1. **What is the new features introduced in Java 5,6,7,8?**

   Java 5: For-each loop (Java 5), Varargs (Java 5), Static Import (Java 5), Autoboxing and Unboxing (Java 5), Enum (Java 5), Covariant Return Type (Java 5), Annotation (Java 5), Generics (Java 5)

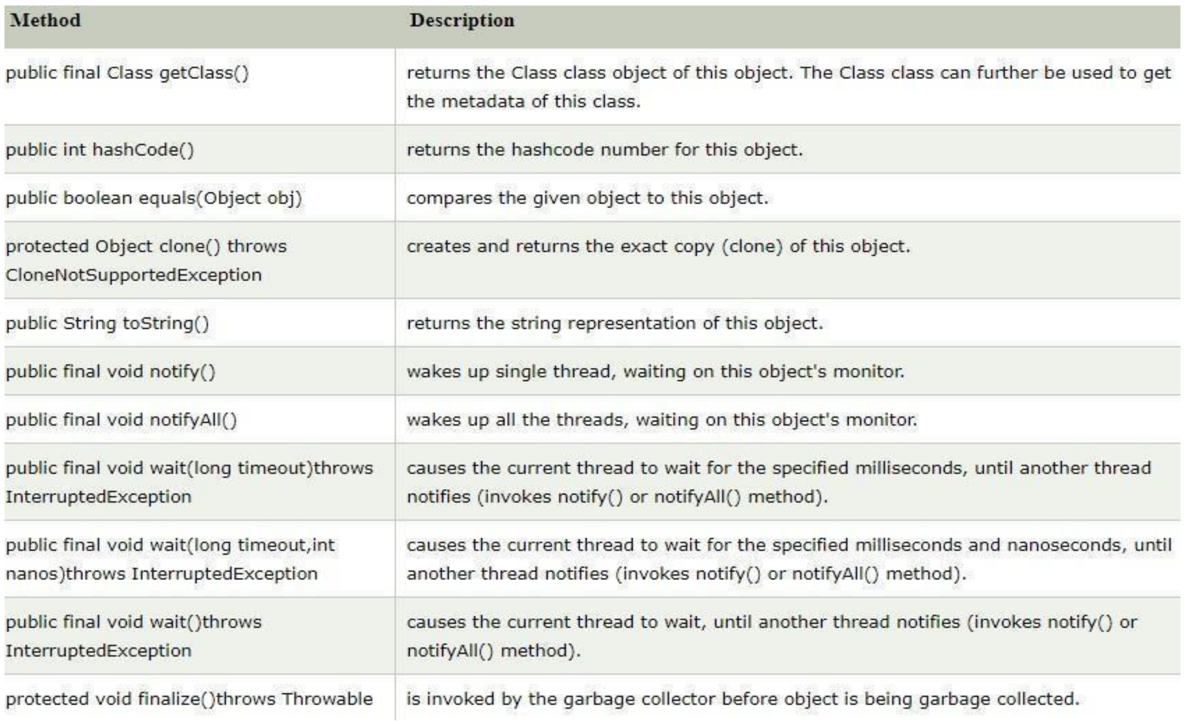
   Java 6: JDBC 4, @Override Annotation

   Java 7: String in Switch, Try with resource, catching multiple exceptions under single catch

   Java 8: Lambda Expressions, Functional Interface, Static and Default Methods, Optional, Method Reference, Date and Time API, Streams.

2. **Explain Object Class Methods?**

   The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.



Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

3. **Why we need the marker interface?**

   Java marker interface are useful if we have information about the class and that information never changes, in such cases, we use marker interface represent to represent the same. Implementing an empty interface tells the compiler to do some operations.

4. Explain about strictfp?

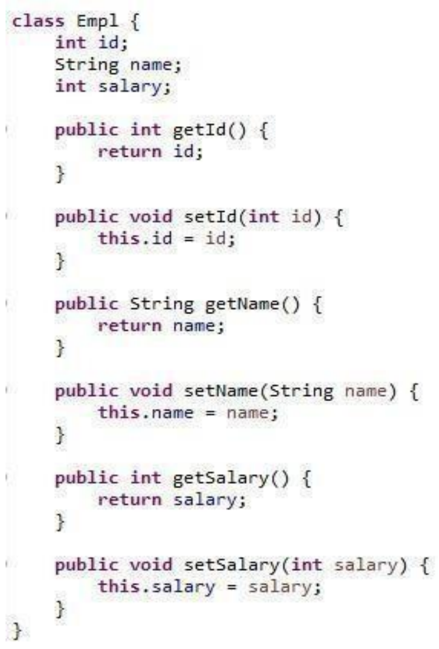
strictfp is used to ensure that floating points operations give the same result on any platform. As floating points precision may vary from one platform to another. strictfp keyword ensures the consistency across the platforms.

strictfp can be applied to class, method or on interfaces but cannot be applied to abstract methods, variable or on constructors.

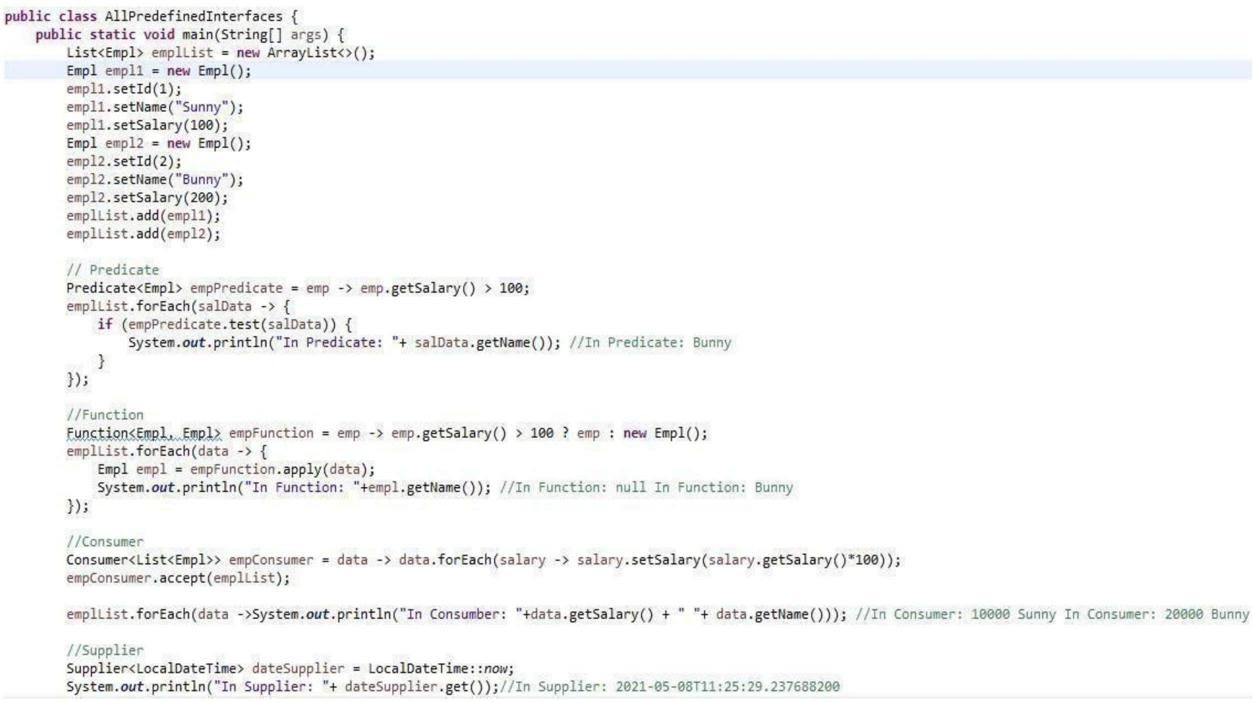
5. Explain the Access modifiers possibility in Inheritance?

Parent Class	**Sub Class**
Public	Public
Protected	Protected, Public
Default	Default, Public and Protected

6. Write a program for Java 8 predefined functional interfaces (Predicate, Function, Consumer, and Supplier)?



```
class Empl {
    int id;
    String name;
    int salary;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getSalary() {
        return salary;
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }
}
```



```java
public class AllPredefinedInterfaces {
    public static void main(String[] args) {
        List<Empl> emplList = new ArrayList<>();
        Empl empl1 = new Empl();
        empl1.setId(1);
        empl1.setName("Sunny");
        empl1.setSalary(100);
        Empl empl2 = new Empl();
        empl2.setId(2);
        empl2.setName("Bunny");
        empl2.setSalary(200);
        emplList.add(empl1);
        emplList.add(empl2);

        // Predicate
        Predicate<Empl> empPredicate = emp -> emp.getSalary() > 100;
        emplList.forEach(salData -> {
            if (empPredicate.test(salData)) {
                System.out.println("In Predicate: "+ salData.getName()); //In Predicate: Bunny
            }
        });

        //Function
        Function<Empl, Empl> empFunction = emp -> emp.getSalary() > 100 ? emp : new Empl();
        emplList.forEach(data -> {
            Empl empl = empFunction.apply(data);
            System.out.println("In Function: "+empl.getName()); //In Function: null In Function: Bunny
        });

        //Consumer
        Consumer<List<Empl>> empConsumer = data -> data.forEach(salary -> salary.setSalary(salary.getSalary()*100));
        empConsumer.accept(emplList);

        emplList.forEach(data ->System.out.println("In Consumber: "+data.getSalary() + " "+ data.getName())); //In Consumer: 10000 Sunny In Consumer: 20000 Bunny

        //Supplier
        Supplier<LocalDateTime> dateSupplier = LocalDateTime::now;
        System.out.println("In Supplier: "+ dateSupplier.get());//In Supplier: 2021-05-08T11:25:29.237688200
```

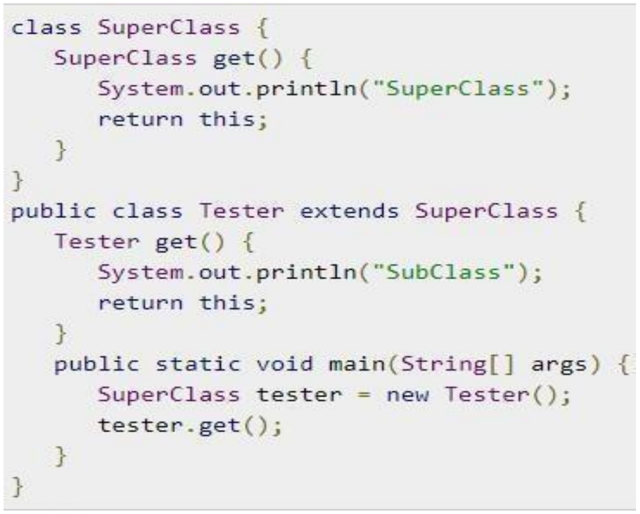
7. How to increase Metaspace size in Java 8?

Java -XX:MetaspaceSize=96M -XX:MaxMetaspaceSize=256m

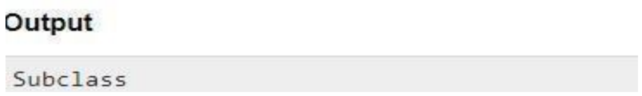
8. What is the use of covariant?

Before JDK 5.0, it was not possible to override a method by changing the return type. When we override a parent class method, the name, argument types and return type of the overriding method in child class has to be exactly same as that of parent class method. Overriding method was said to be invariant with respect to return type.

Java 5.0 onwards it is possible to have different return type for an overriding method in child class, but child's return type should be sub-type of parent's return type. Covariant return type works only for non-primitive return types.



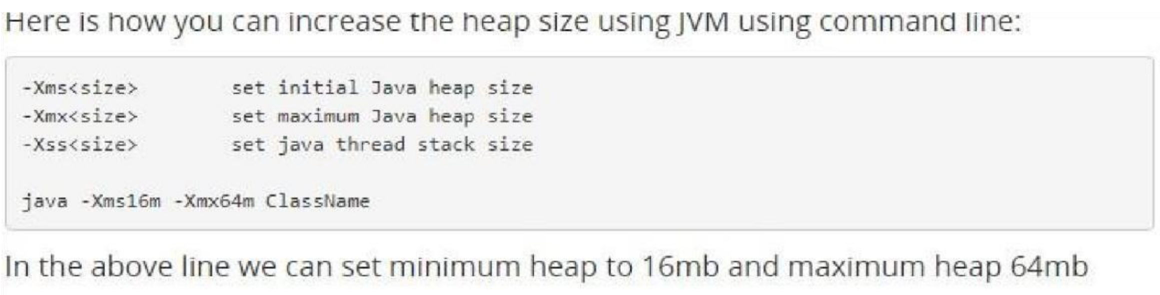
```java
class SuperClass {
   SuperClass get() {
      System.out.println("SuperClass");
      return this;
   }
}
public class Tester extends SuperClass {
   Tester get() {
      System.out.println("SubClass");
      return this;
   }
   public static void main(String[] args) {
      SuperClass tester = new Tester();
      tester.get();
   }
}
```



Output

```
Subclass
```

9. How to increase the heap size?



Here is how you can increase the heap size using JVM using command line:

```
-Xms<size>        set initial Java heap size
-Xmx<size>        set maximum Java heap size
-Xss<size>        set java thread stack size

java -Xms16m -Xmx64m ClassName
```

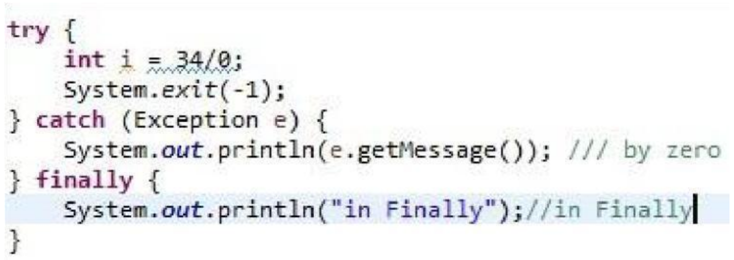
In the above line we can set minimum heap to 16mb and maximum heap 64mb

10. Difference between Class.forName() and ClassLoader.loadClass()?

Class.forName(): load and initialize the class. In class loader subsystem it executes all the three phases i.e. load, link, and initialize phases.
ClassLoader.loadClass(): behavior, which delays initialization until the class is used for the first time. In class loader subsystem it executes only two phases i.e. load and link phases.

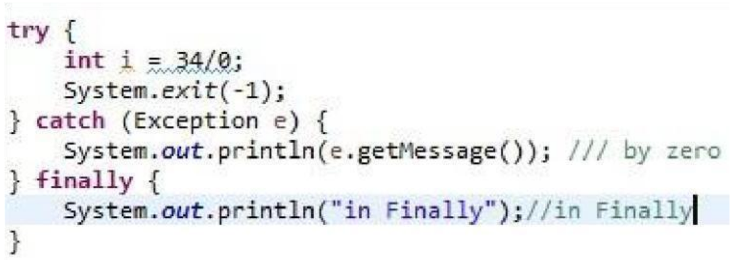
11. Can we add System.exit() method in try block?

Yes we can add.


```
try {
    int i = 34/0;
    System.exit(-1);
} catch (Exception e) {
    System.out.println(e.getMessage()); /// by zero
} finally {
    System.out.println("in Finally");//in Finally
}
```

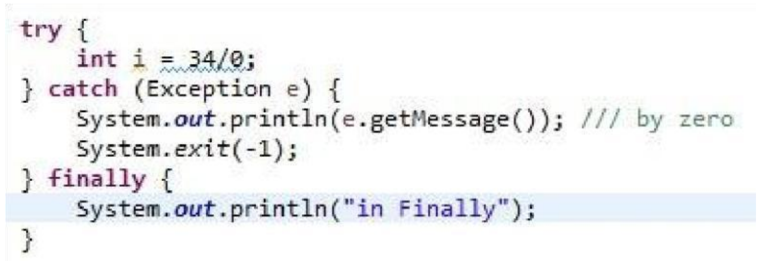
12. If we add the System.exit() in try block is finally block will skip?

No. It will not skip the finally block,


```
try {
    int i = 34/0;
    System.exit(-1);
} catch (Exception e) {
    System.out.println(e.getMessage()); /// by zero
} finally {
    System.out.println("in Finally");//in Finally
}
```

13. How to skip the finally block execution?

By adding the System.exit() in catch block.


```
try {
    int i = 34/0;
} catch (Exception e) {
    System.out.println(e.getMessage()); /// by zero
    System.exit(-1);
} finally {
    System.out.println("in Finally");
}
```

14. Explain more about WeakHashMap and IdentityHashMap?

WeakHashMap: WeakHashMap is an implementation of the Map interface that stores only weak references to its keys. Storing only weak references allows a key-value pair to be garbagecollected when its key is no longer referenced outside of the WeakHashMap.

This class provides the easiest way to harness the power of weak references. It is useful for implementing "registry-like" data structures, where the utility of an entry vanishes when its key is no longer reachable by any thread.

IdentityHashMap: It is similar to HashMap except that it uses reference equality when comparing elements. It is mandates the use of the equals() method when comparing objects. It is best suitable in Serialization and cloning the objects and getting the proxy of an object for debugging.

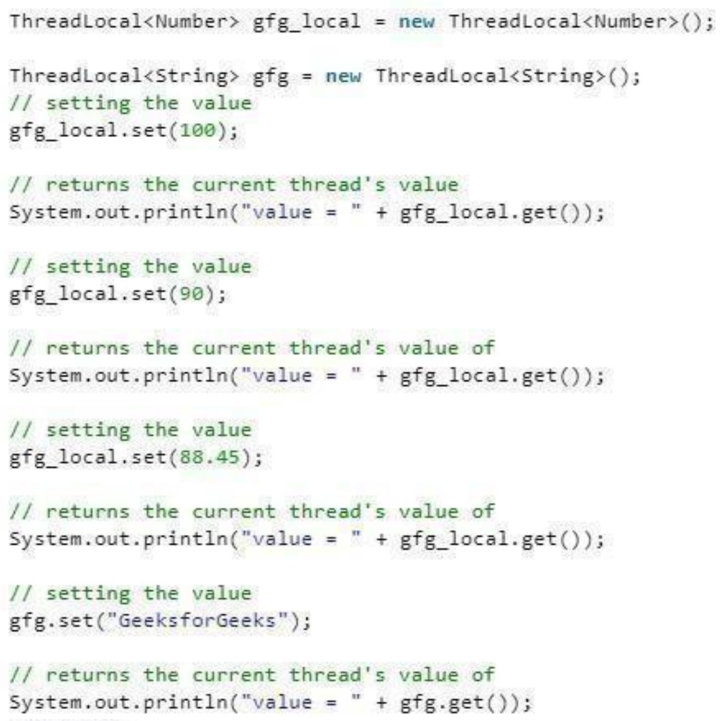
15. Explain about ThreadLocal?

ThreadLocal in Java is another way to achieve thread-safety. In thread local, you can set any object and this object will be local and global to the specific thread which is accessing this object. Java ThreadLocal class provides thread-local variables. It enables you to create variables that can only be read and write by the same thread. If two threads are executing the same code and that code has a reference to a ThreadLocal variable then the two threads can't see the local variable of each other. **As part of ThreadLocal class we have three methods,**

Set

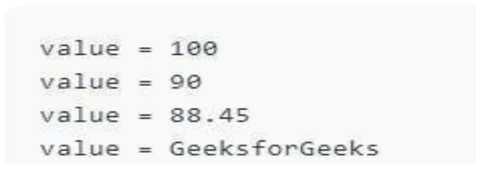
Get

Remove



```
ThreadLocal<Number> gfg_local = new ThreadLocal<Number>();

ThreadLocal<String> gfg = new ThreadLocal<String>();
// setting the value
gfg_local.set(100);

// returns the current thread's value
System.out.println("value = " + gfg_local.get());

// setting the value
gfg_local.set(90);

// returns the current thread's value of
System.out.println("value = " + gfg_local.get());

// setting the value
gfg_local.set(88.45);

// returns the current thread's value of
System.out.println("value = " + gfg_local.get());

// setting the value
gfg.set("GeeksforGeeks");

// returns the current thread's value of
System.out.println("value = " + gfg.get());
```

Output:



```
value = 100
value = 90
value = 88.45
value = GeeksforGeeks
```

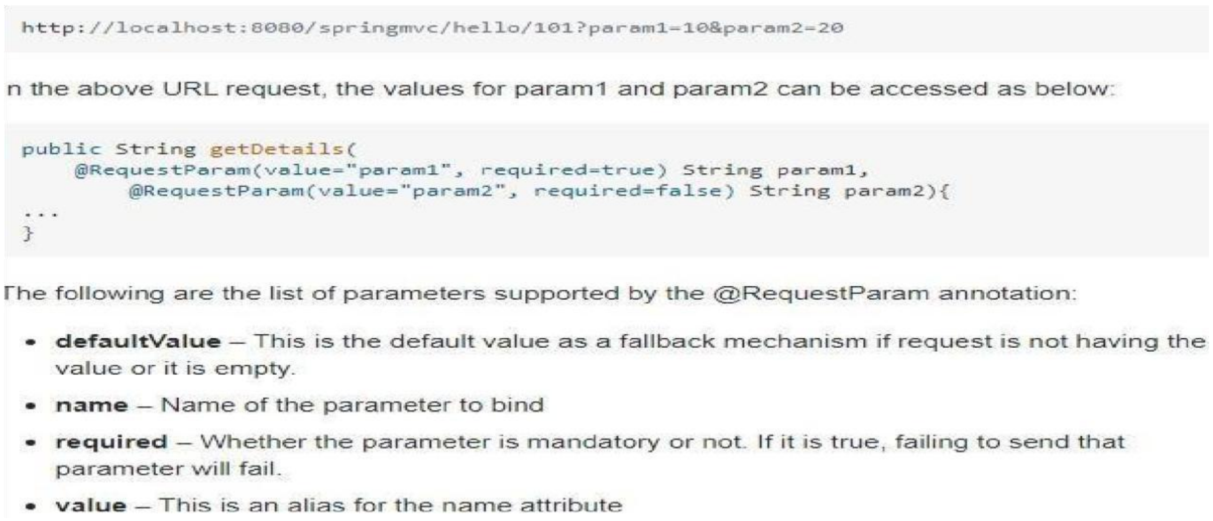
16. Explain abut Thread Scheduler?

Thread scheduler in java is the part of the JVM that decides which thread should run. There is no guarantee that which runnable thread will be chosen to run by the thread scheduler. Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

17. Difference between @RequestParam, @QueryParam, @PathParam and @PathVariable?

@RequestParam annotation used for accessing the query parameter values from the request. @RequestParam is more useful on a traditional web application where data is mostly passed in the query parameters. @RequestParam annotation can specify default values if a query parameter is not present or empty by using a defaultValue attribute, provided the required attribute is false.



```
http://localhost:8080/springmvc/hello/101?param1=10&param2=20
```

n the above URL request, the values for param1 and param2 can be accessed as below:

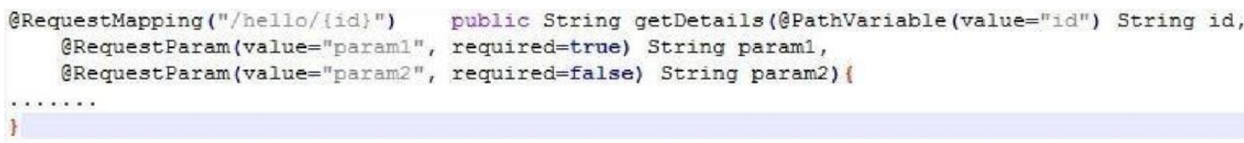
```
public String getDetails(
    @RequestParam(value="param1", required=true) String param1,
        @RequestParam(value="param2", required=false) String param2){
...
}
```

The following are the list of parameters supported by the @RequestParam annotation:

- **defaultValue** – This is the default value as a fallback mechanism if request is not having the value or it is empty.
- **name** – Name of the parameter to bind
- **required** – Whether the parameter is mandatory or not. If it is true, failing to send that parameter will fail.
- **value** – This is an alias for the name attribute

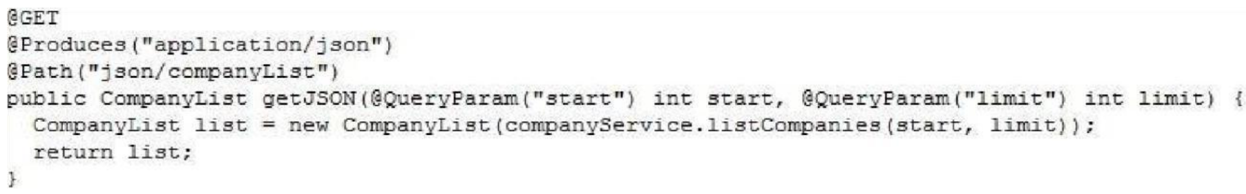
@PathVariable: Identifies the pattern that is used in the URI for the incoming request. It is best suitable in Spring MVC application. We can use it in Rest service also.

http://localhost:8080/springmvc/hello/101?param1=10&param2=20



```
@RequestMapping("/hello/{id}")    public String getDetails(@PathVariable(value="id") String id,
    @RequestParam(value="param1", required=true) String param1,
    @RequestParam(value="param2", required=false) String param2){
.......
}
```

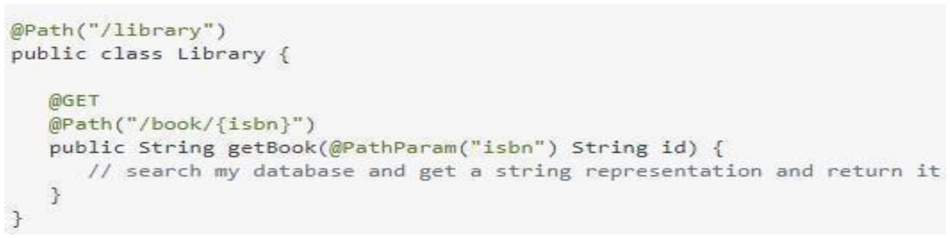
@QueryParam: Request parameters in query string can be accessed using @QueryParam annotation.



```
@GET
@Produces("application/json")
@Path("json/companyList")
public CompanyList getJSON(@QueryParam("start") int start, @QueryParam("limit") int limit) {
  CompanyList list = new CompanyList(companyService.listCompanies(start, limit));
  return list;
}
```

@PathParam: It will injects value from URI to your method input parameters.

http://localhost:8080/books/1234



```
@Path("/library")
public class Library {

   @GET
   @Path("/book/{isbn}")
   public String getBook(@PathParam("isbn") String id) {
      // search my database and get a string representation and return it
   }
}
```

18. Explain about idempotent methods?

Idempotent operations produce the same result even when the operation is repeated many times. The result of the 2nd, 3rd, and 1,000th repeat of the operation will return exactly the same result as the 1st time.

Below are the idempotent methods

Get

Put

Head

Delete

19. Benefits of encapsulation?

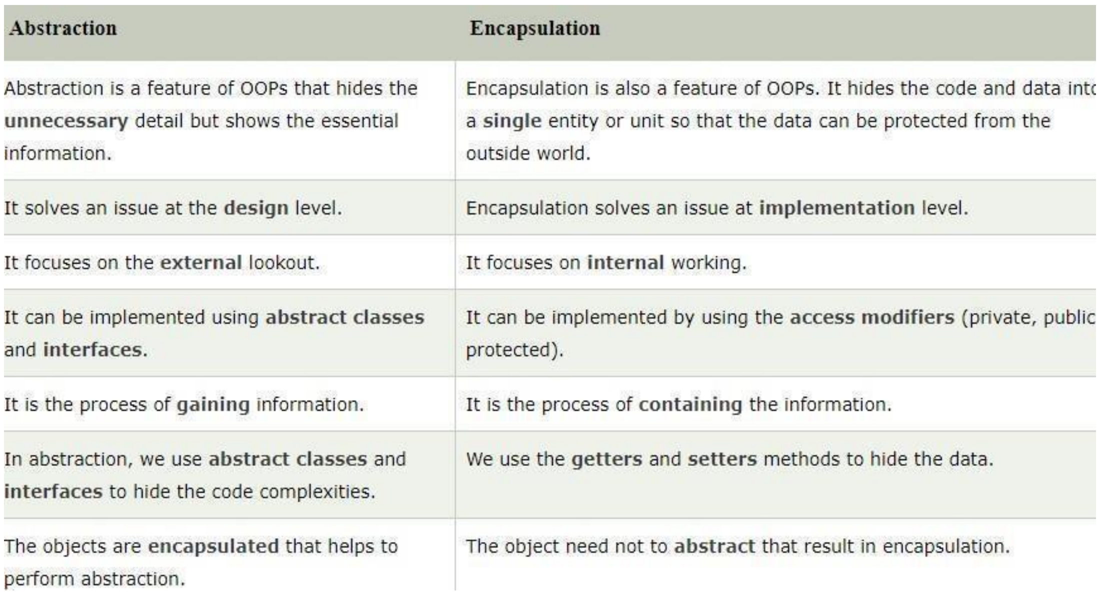
A class can have total control over what is stored in its fields.

Data Hiding

Reusability

Easy to test

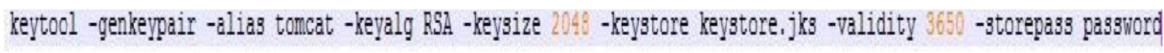
20. Abstraction vs Encapsulation?



Abstraction	Encapsulation
Abstraction is a feature of OOPs that hides the **unnecessary** detail but shows the essential information.	Encapsulation is also a feature of OOPs. It hides the code and data int a **single** entity or unit so that the data can be protected from the outside world.
It solves an issue at the **design** level.	Encapsulation solves an issue at **implementation** level.
It focuses on the **external** lookout.	It focuses on **internal** working.
It can be implemented using **abstract classes** and **interfaces.**	It can be implemented by using the **access modifiers** (private, public protected).
It is the process of **gaining** information.	It is the process of **containing** the information.
In abstraction, we use **abstract classes** and **interfaces** to hide the code complexities.	We use the **getters** and **setters** methods to hide the data.
The objects are **encapsulated** that helps to perform abstraction.	The object need not to **abstract** that result in encapsulation.

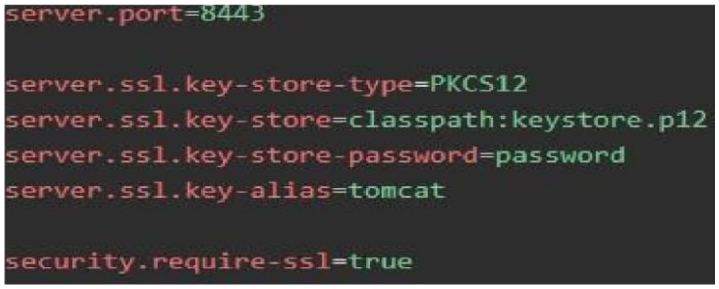
21. How to convert from HTTP to HTTPS?

Generate or buy the SSL certificate. We can generate the self-certificate by using keytool command,



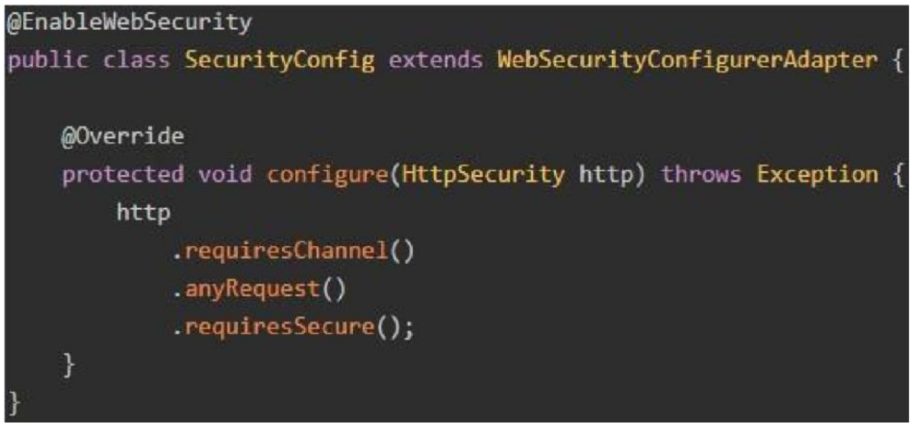
```
keytool -genkeypair -alias tomcat -keyalg RSA -keysize 2048 -keystore keystore.jks -validity 3650 -storepass password
```

After certificate generation add all the certificate details in application.properties



```
server.port=8443

server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:keystore.p12
server.ssl.key-store-password=password
server.ssl.key-alias=tomcat

security.require-ssl=true
```

Block the HTTP request by using Spring Security



```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .requiresChannel()
            .anyRequest()
            .requiresSecure();
    }
}
```

If required add the code to redirect HTTP request to HTTPS.

22. Difference java.util.date and SQL date?

SQL Date just represent DATE without time information while java.util.Date represents both Date and Time information.

23. Can we have equals without hashcode?

Yes, you can only implement equals() method without implementing hashcode() method.

But standard practice says that you should implement both of them and for the equal object the hashcode should be the same.

24. Can we have hashcode without equals?

Yes, you can only implement hashcode() method without implementing equals() method.

But standard practice says that you should implement both of them and for the equal object the hashcode should be the same.

25. Why we need to override the equals and hashcode?

HashMap, HashTable and HashSet use the hashcode value of an object to find out how the object would be stored in the collection, and subsequently hashcode is used to help locate the object in the collection. Hashing retrieval involves:

First, find out the right bucket using hashCode().

Secondly, search the bucket for the right element using equals()

26. Does ArrayList used hashcode() and equals()?

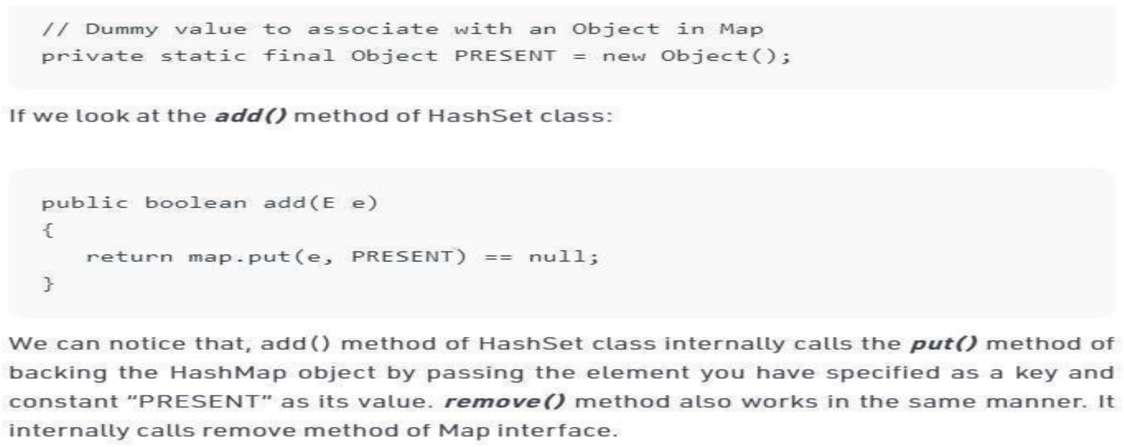
The hashCode of ArrayList is a function of the hashCode of all the elements stored in the ArrayList, so it doesn't change when the capacity changes, it changes whenever you add or remove an element or mutate one of the elements in a way that changes its hashCode.

27. Is it possible to have Hashmap without hashcode and equals?

If you are maintaining the key as object then it is possible. Any java object that does not define its own equals and hashcode methods inherit the default equals and hashcode methods on java.lang.Object. These default implementation are based on object reference equality and not on logically equality. Since you have called get with the same object reference the object can be returned from the map.

28. HashSet internally uses HashMap but HashSet does not store key value pair, In internally how its handled?

Actually the value we insert in HashSet acts as a key to the map Object and for its value, java uses a constant variable. So in key-value pair, all the values will be the same.



```
// Dummy value to associate with an Object in Map
private static final Object PRESENT = new Object();
```

If we look at the ***add()*** method of HashSet class:

```
public boolean add(E e)
{
    return map.put(e, PRESENT) == null;
}
```

We can notice that, add() method of HashSet class internally calls the ***put()*** method of backing the HashMap object by passing the element you have specified as a key and constant "PRESENT" as its value. ***remove()*** method also works in the same manner. It internally calls remove method of Map interface.

29. Can we make Restful web services stateful?

Yes. Rest engages in state transfer and to make them stateful, we can use client side or db persisted session state, and transfer them across web service invocations as an attribute in either the header or a method parameter.

30. Explain the techniques to tune the database?

Apply the Normal Forms

While using SELECT statement, only fetch whatever information is required and avoid using * in your SELECT queries because it would load the system unnecessarily.
Create your indexes carefully on all the tables where you have frequent search operations. Avoid index on the tables where you have less number of search operations and more number of insert and update operations.

For queries that are executed on a regular basis, try to use procedures. A procedure is a potentially large group of SQL statements.

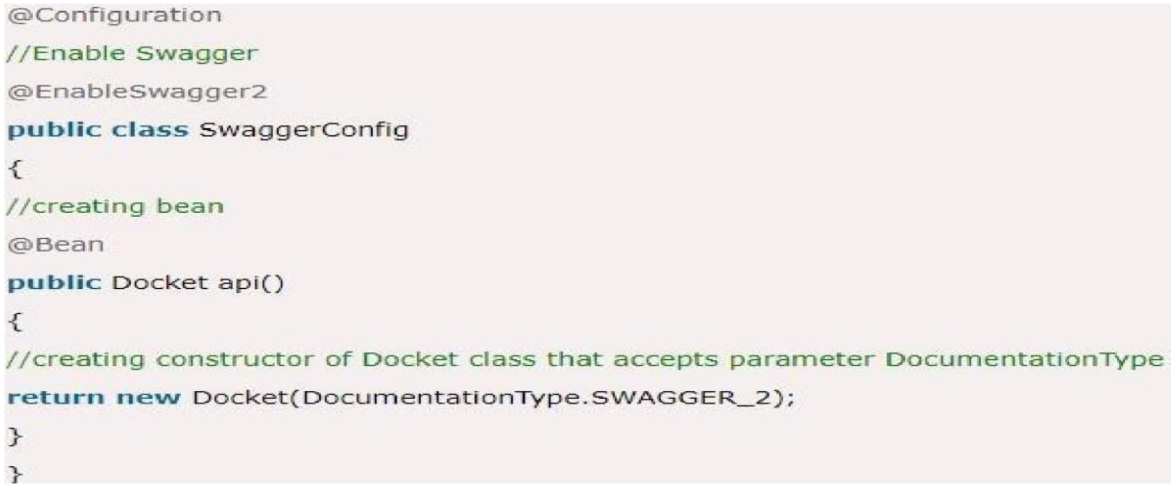
Avoid the loops.

31. Explain about Swagger?

A Swagger is an open-source tool. It is the most popular API documentation format for RESTful Web Services. It provides both JSON and UI support. JSON can be used as a machine-readable format, and Swagger-UI is for visual display.

Currently we are using Swagger 2. Latest version of swagger is 2.2.1

To enable Swagger we need to add the dependent jars and Swagger configuration class,

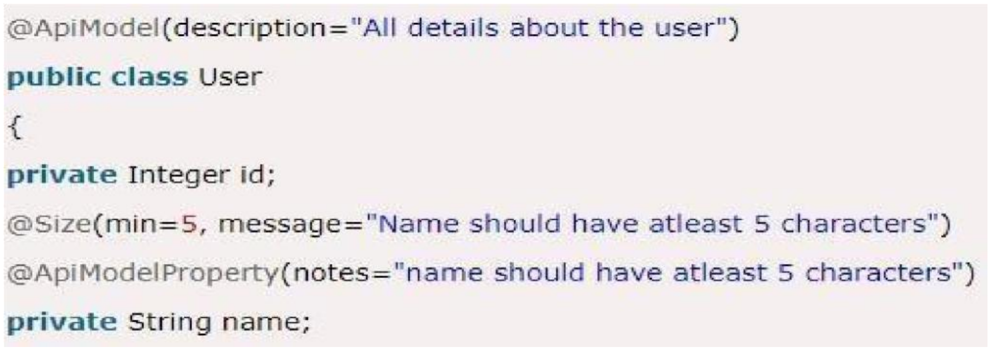


```
@Configuration
//Enable Swagger
@EnableSwagger2
public class SwaggerConfig
{
//creating bean
@Bean
public Docket api()
{
//creating constructor of Docket class that accepts parameter DocumentationType
return new Docket(DocumentationType.SWAGGER_2);
}
}
```

JSON format Documentation: http://localhost:8080/v2/api-docs

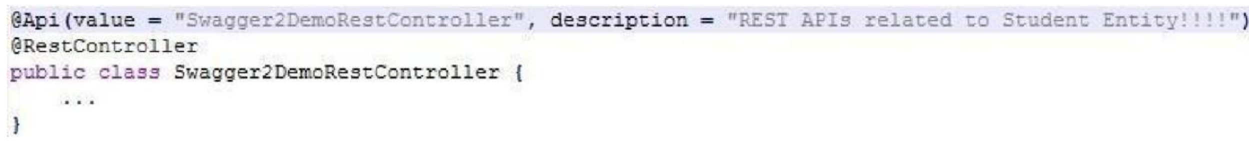
Swagger UI: http://localhost:8080/swagger-ui.html **Swagger Annotations:**

@ApiModel: It provides additional information about Swagger Models.
@ApiModelProperty: It allows controlling swagger-specific definitions such as values, and additional notes.



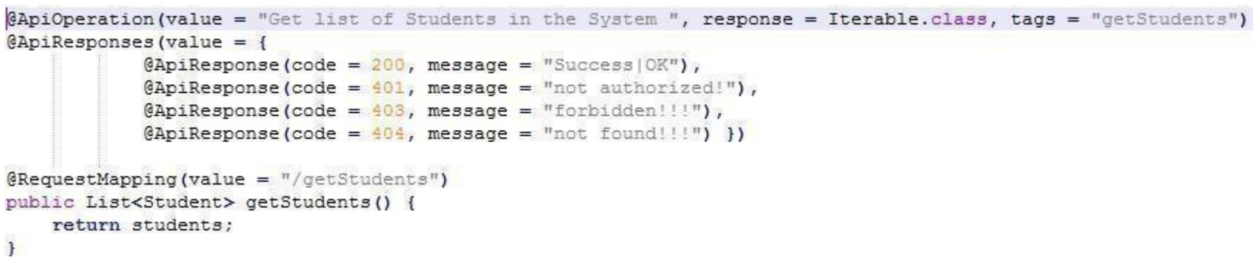
```
@ApiModel(description="All details about the user")
public class User
{
private Integer id;
@Size(min=5, message="Name should have atleast 5 characters")
@ApiModelProperty(notes="name should have atleast 5 characters")
private String name;
```

@Api – We can add this Annotation to the controller to add basic information regarding the controller.



```
@Api(value = "Swagger2DemoRestController", description = "REST APIs related to Student Entity!!!!")
@RestController
public class Swagger2DemoRestController {
    ...
}
```

@ApiOperation and @ApiResponses – We can add these annotations to any rest method in the controller to add basic information related to that method.

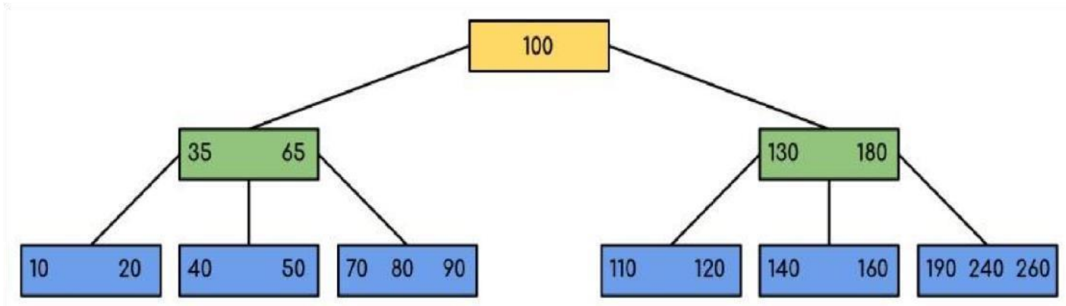


```
@ApiOperation(value = "Get list of Students in the System ", response = Iterable.class, tags = "getStudents")
@ApiResponses(value = {
            @ApiResponse(code = 200, message = "Success|OK"),
            @ApiResponse(code = 401, message = "not authorized!"),
            @ApiResponse(code = 403, message = "forbidden!!!"),
            @ApiResponse(code = 404, message = "not found!!!") })

@RequestMapping(value = "/getStudents")
public List<Student> getStudents() {
    return students;
}
```

32. How HashMap performance increased in java8?

Hash collisions have negative impact on the lookup time of HashMap. When multiple keys end up in the same bucket, then values along with their keys are placed in a linked list. In case of retrieval, linked list has to be traversed to get the entry. In worst case scenario, when all keys are mapped to the same bucket, the lookup time of HashMap increases.

To address this issue, Java 8 hash elements **use balanced trees (B Tree)** instead of linked lists after a certain threshold is reached. Which means HashMap starts with storing Entry objects in a linked list but after the number of items in a hash becomes larger than a certain threshold. The hash will change from using a linked list to a balanced tree.



The alternative String hash function added in Java 7 has been removed.

33. Give the best places for Immutable classes?

Multithreading environment

HashMap Key

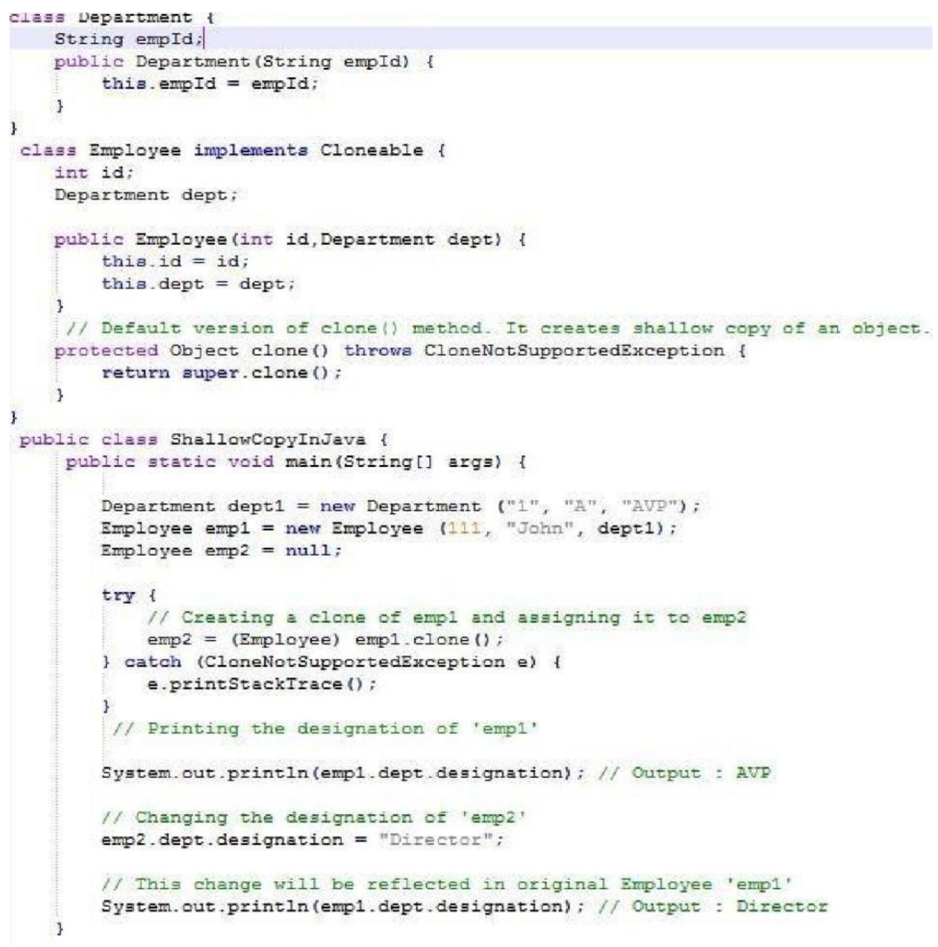
34. What is meant by cloning and explain different types of cloning?

Cloning is the process of getting the exact copy of an object. The clone() method of Object class is used to clone an object.

The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates CloneNotSupportedException.

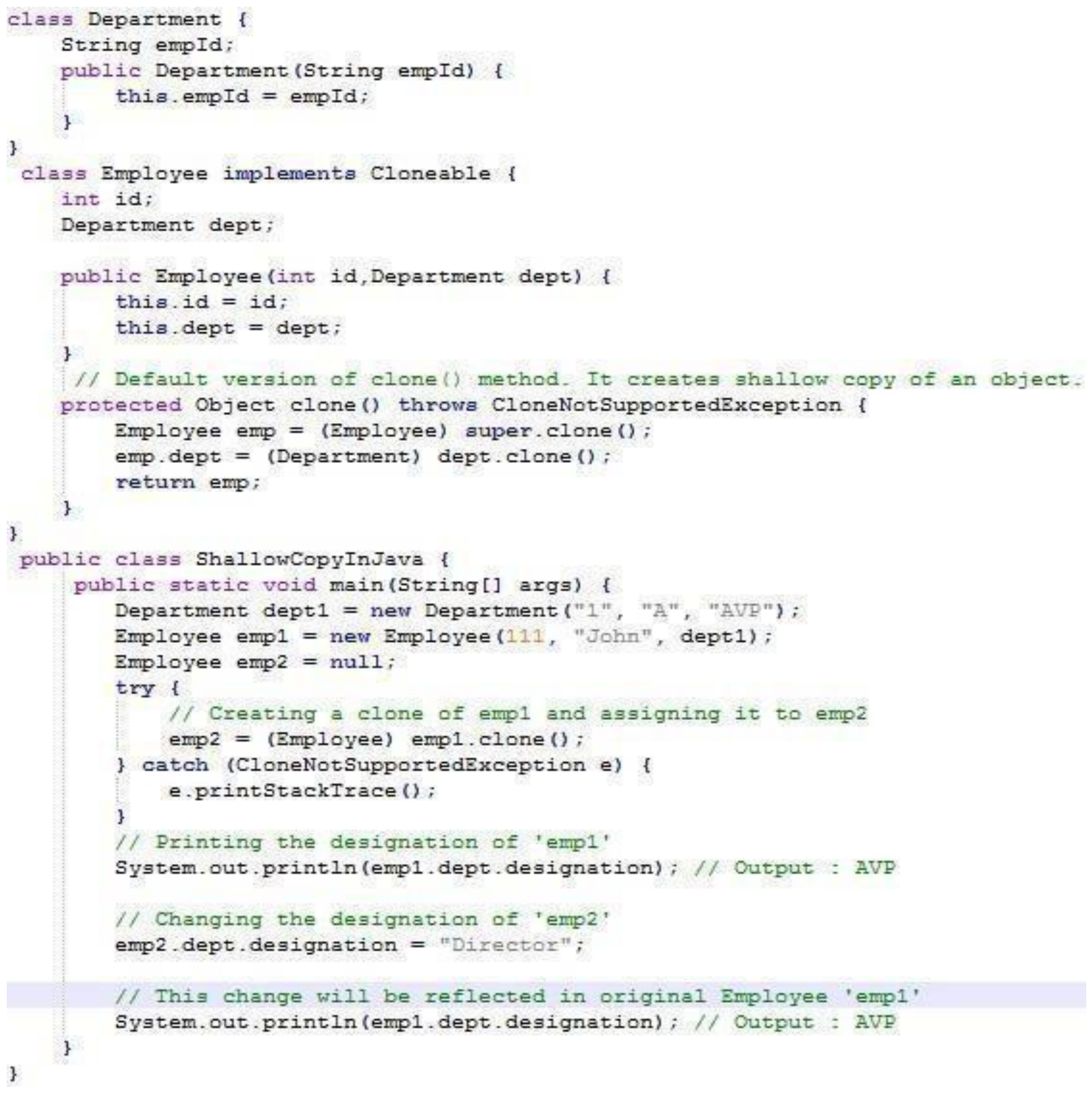
Below are the types of cloning,

Shallow Cloning: A shallow copy of an object copies the 'main' object, but doesn't copy the inner objects. If we make changes in shallow copy then changes will get reflected in the source object. Both instances are not independent.



```java
class Department {
    String empId;
    public Department(String empId) {
        this.empId = empId;
    }
}
 class Employee implements Cloneable {
    int id;
    Department dept;

    public Employee(int id,Department dept) {
        this.id = id;
        this.dept = dept;
    }
    // Default version of clone() method. It creates shallow copy of an object.
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
 public class ShallowCopyInJava {
     public static void main(String[] args) {

        Department dept1 = new Department ("1", "A", "AVP");
        Employee emp1 = new Employee (111, "John", dept1);
        Employee emp2 = null;

        try {
            // Creating a clone of emp1 and assigning it to emp2
            emp2 = (Employee) emp1.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
         // Printing the designation of 'emp1'

        System.out.println(emp1.dept.designation); // Output : AVP

        // Changing the designation of 'emp2'
        emp2.dept.designation = "Director";

        // This change will be reflected in original Employee 'emp1'
        System.out.println(emp1.dept.designation); // Output : Director
    }
```

Deep Cloning: A Deep copy of an object copies the 'main' object and its inner objects also. If we make changes in deep copy then changes will not reflected in the source object. Both instances are independent.



```
class Department {
    String empId;
    public Department(String empId) {
        this.empId = empId;
    }
}
 class Employee implements Cloneable {
    int id;
    Department dept;

    public Employee(int id,Department dept) {
        this.id = id;
        this.dept = dept;
    }
     // Default version of clone() method. It creates shallow copy of an object.
    protected Object clone() throws CloneNotSupportedException {
        Employee emp = (Employee) super.clone();
        emp.dept = (Department) dept.clone();
        return emp;
    }
}
 public class ShallowCopyInJava {
     public static void main(String[] args) {
        Department dept1 = new Department("1", "A", "AVP");
        Employee emp1 = new Employee(111, "John", dept1);
        Employee emp2 = null;
        try {
            // Creating a clone of emp1 and assigning it to emp2
            emp2 = (Employee) emp1.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        // Printing the designation of 'emp1'
        System.out.println(emp1.dept.designation); // Output : AVP

        // Changing the designation of 'emp2'
        emp2.dept.designation = "Director";

        // This change will be reflected in original Employee 'emp1'
        System.out.println(emp1.dept.designation); // Output : AVP
    }
}
```

35. How to convert Monolith to Micro Services?

- **Split the Front end and Backend**

  Presentation Layer

  Business Layer

  Data-access Layer

- **Analyze the database and make it independent tables.**

36. How to avoid the Dead Lock?

The deadlock is a situation when two or more threads try to access the same object that is acquired by another thread. Since the threads wait for releasing the object, the condition is known as deadlock.

We can avoid the dead lock by using below techniques,

Use Thread.join() method

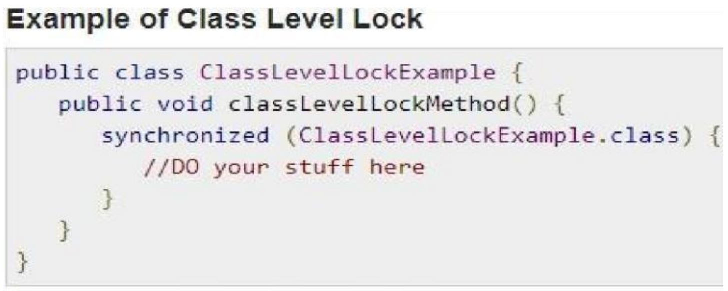
Apply the timeout on each thread

Apply the Thread priorities

37. Difference between Class level and Object level locking?

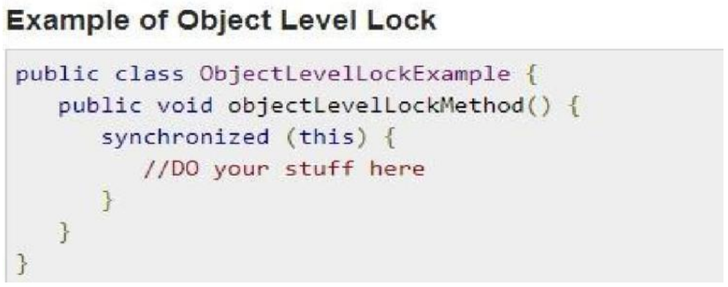
Object Level Locks − It can be used when you want non-static method or non-static block of the code should be accessed by only one thread.

Class Level locks − It can be used when we want to prevent multiple threads to enter the synchronized block in any of all available instances on runtime. It should always be used to make static data thread safe.



Example of Class Level Lock

```
public class ClassLevelLockExample {
   public void classLevelLockMethod() {
      synchronized (ClassLevelLockExample.class) {
         //DO your stuff here
      }
   }
}
```



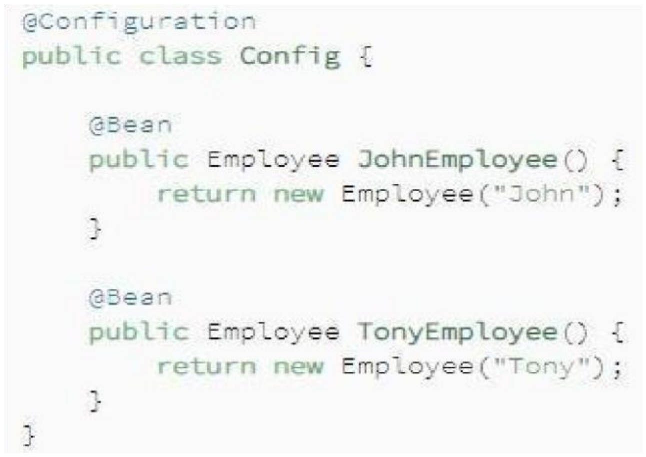
Example of Object Level Lock

```
public class ObjectLevelLockExample {
   public void objectLevelLockMethod() {
      synchronized (this) {
         //DO your stuff here
      }
   }
}
```

38. What is the use of @Primary annotation?

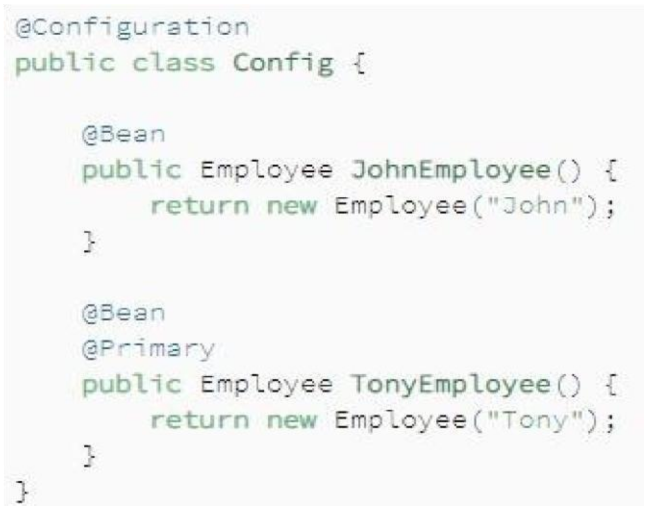
@Primary to give higher preference to a bean when there are multiple beans of the same type.

In some cases, we need to register more than one bean of the same type.



```
@Configuration
public class Config {

    @Bean
    public Employee JohnEmployee() {
        return new Employee("John");
    }

    @Bean
    public Employee TonyEmployee() {
        return new Employee("Tony");
    }
}
```

To access beans with the same type we usually use @Qualifier("beanName") annotation. We apply it at the injection point along with @Autowired. In our case, we select the beans at the configuration phase so @Qualifier can't be applied here.



```
@Configuration
public class Config {

    @Bean
    public Employee JohnEmployee() {
        return new Employee("John");
    }

    @Bean
    @Primary
    public Employee TonyEmployee() {
        return new Employee("Tony");
    }
}
```

39. Difference between Filter and Interceptor?

Filter: A filter as the name suggests is a Java class executed by the servlet container for each incoming HTTP request and for each http response. This way, is possible to manage HTTP incoming requests before them reach the resource.

Interceptor: Spring Interceptors are similar to Servlet Filters but they acts in Spring Context so are many powerful to manage HTTP Request and Response but they can implement more sophisticated behavior because can access to all Spring context. In the life cycle of an action, interceptors can be called multiple times, while filters can only be called once when the container is initialized. The interceptor can get the bean s in the IOC container, but the filter can't. This is very important. Inject a service into the interceptor to call the business logic. The interceptor is wrapped in a filter.

40. What is the use of load-on-startup in servlet?

The load-on-startup element of servlet in web.xml is used to load the servlet at the time of deploying the project or server start. So it saves time for the response of first request.

If you pass the positive value, the lower integer value servlet will be loaded before the higher integer value servlet. In other words, container loads the servlets in ascending integer value. The 0 value will be loaded first then 1, 2, 3 and so on.

If you pass the negative value, servlet will be loaded at request time, at first request.

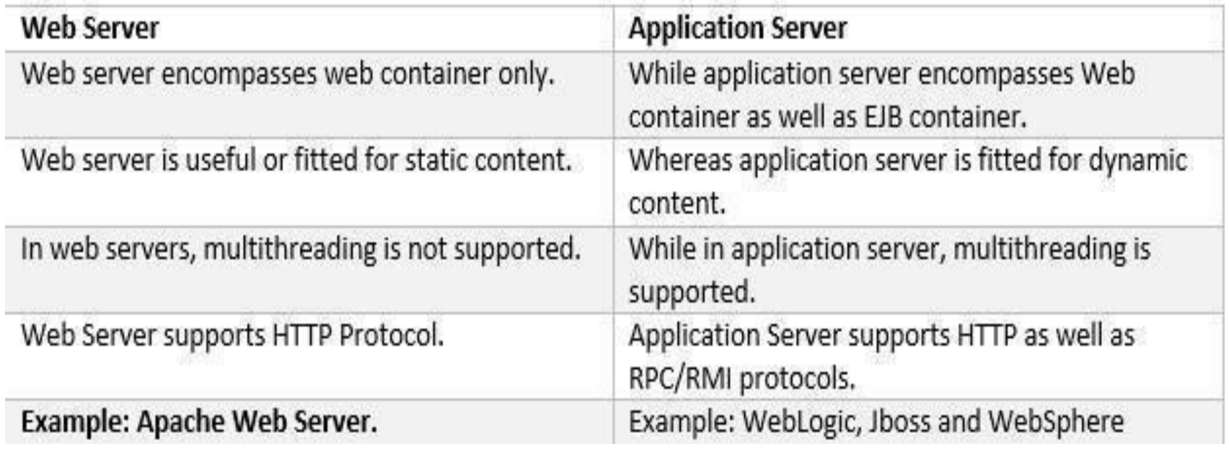
41. Difference between Jar, War and Ear?

Jar: JAR file is a combination of all these files as a single compressed file. It consists of Java source codes, XML based configuration data, manifest file, JSON based data files, etc.

War: A WAR file contains the files of a web project. It can have servlet, JSP, XML, HTML, CSS and JavaScript files. These files can be deployed on servlet/JSP container. The WAR files are inside the WEB-INF folder of the project. As a WAR file combines all the files into a single unit, it takes less time to exchange a file from the client to server.

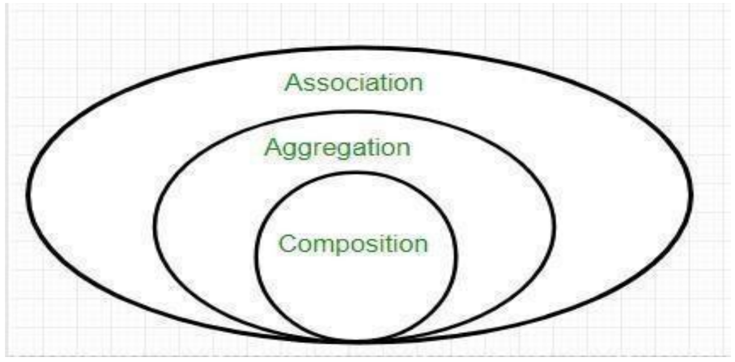
Ear: An EAR file is a Java EE file. It packs one or more modules into one archive. Also, this helps to deploy various modules onto an application server simultaneously and coherently. EAR file consists of deployment descriptors that describe how to deploy the modules. These deployment descriptors are XML files. Moreover, applications such as Ant, Maven, and, Gradle help to build EAR files.

42. Difference between Web Server and Application Server?



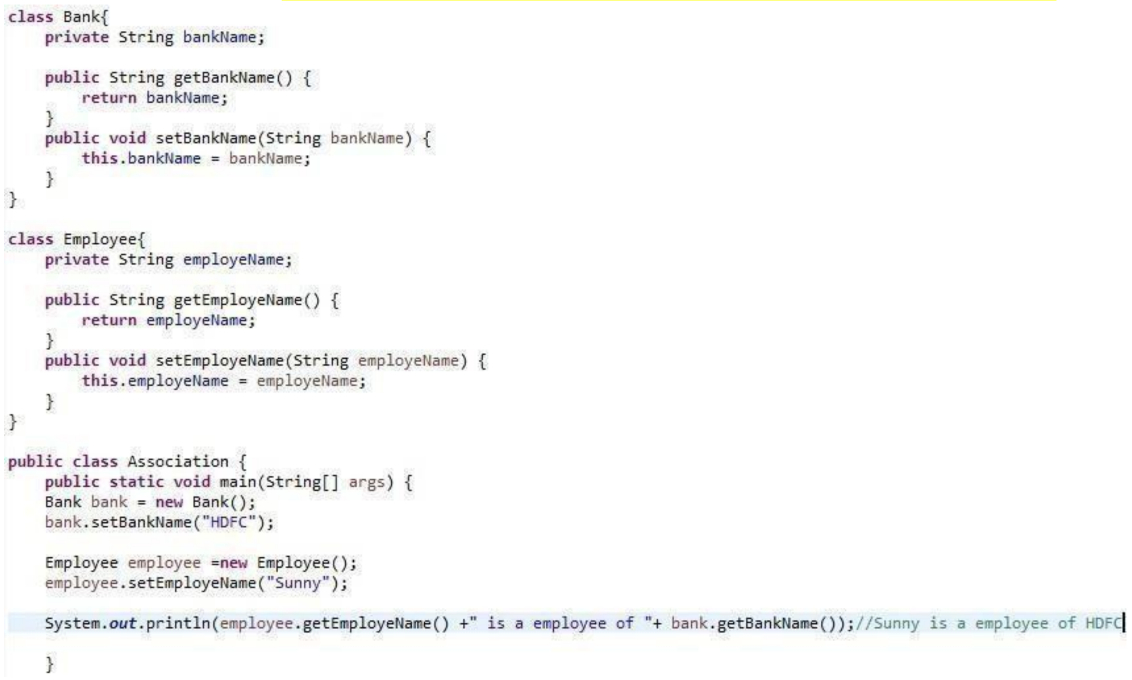
Web Server	Application Server
Web server encompasses web container only.	While application server encompasses Web container as well as EJB container.
Web server is useful or fitted for static content.	Whereas application server is fitted for dynamic content.
In web servers, multithreading is not supported.	While in application server, multithreading is supported.
Web Server supports HTTP Protocol.	Application Server supports HTTP as well as RPC/RMI protocols.
Example: Apache Web Server.	Example: WebLogic, Jboss and WebSphere

43. Difference between Association, Aggregation and Composition?



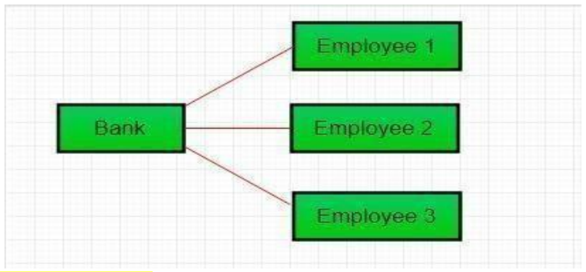
Association: Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one and many-to-many.

In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object**. Composition and Aggregation are the two forms of association**.



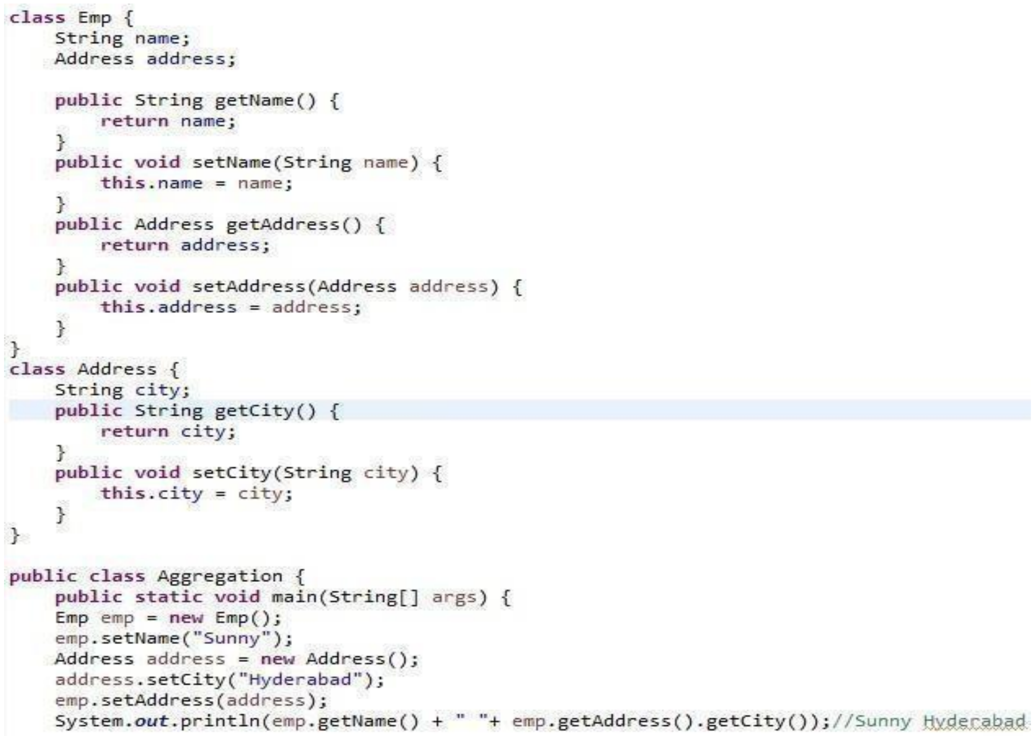
```
class Bank{
    private String bankName;

    public String getBankName() {
        return bankName;
    }
    public void setBankName(String bankName) {
        this.bankName = bankName;
    }
}

class Employee{
    private String employeName;

    public String getEmployeName() {
        return employeName;
    }
    public void setEmployeName(String employeName) {
        this.employeName = employeName;
    }
}

public class Association {
    public static void main(String[] args) {
    Bank bank = new Bank();
    bank.setBankName("HDFC");

    Employee employee =new Employee();
    employee.setEmployeName("Sunny");

    System.out.println(employee.getEmployeName() +" is a employee of "+ bank.getBankName());//Sunny is a employee of HDFC

    }
```

In above example two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, so it is a one-to-many relationship.



Aggregation: It represents **Has-A** relationship. It is a unidirectional association i.e. a one way relationship. For example, Student class can have reference of Address class but vice versa does not make sense.

In Aggregation, both the entries can survive individually which means ending one entity will not affect the other entity



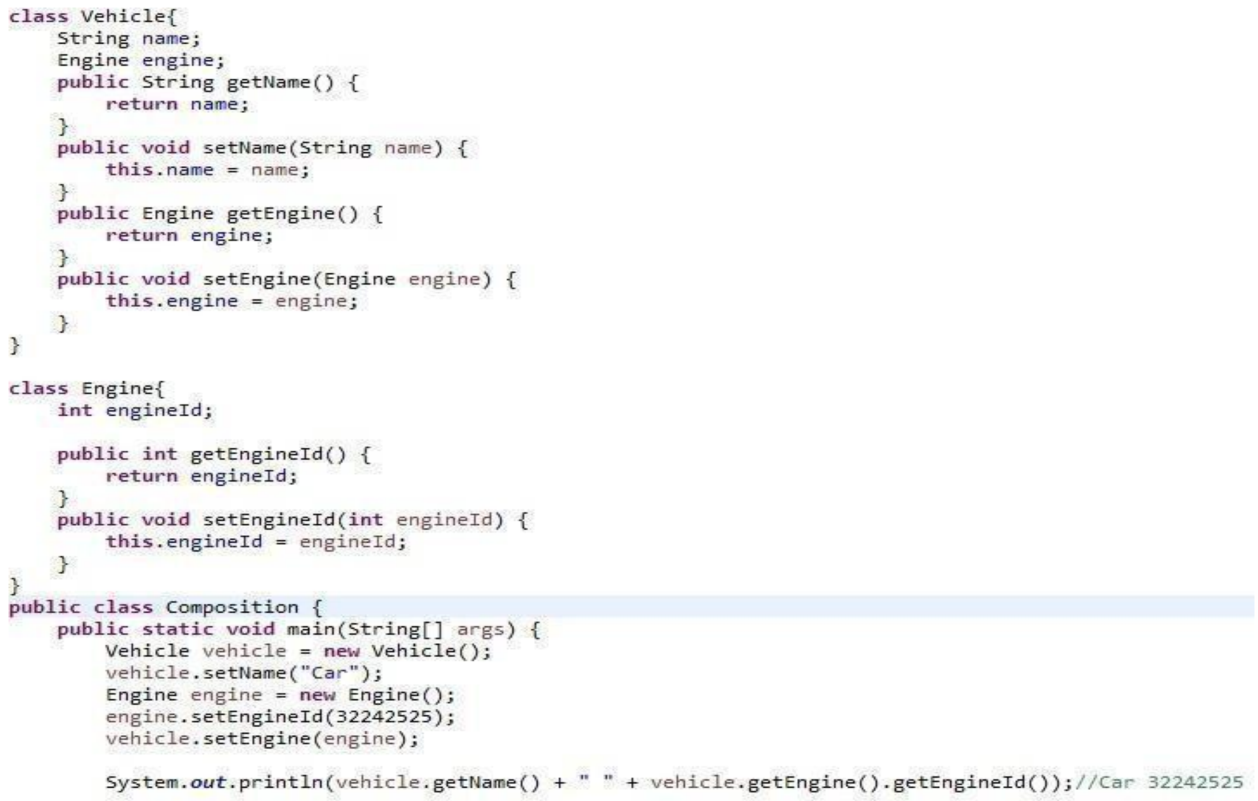
```
class Emp {
    String name;
    Address address;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        this.address = address;
    }
}
class Address {
    String city;
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
}

public class Aggregation {
    public static void main(String[] args) {
    Emp emp = new Emp();
    emp.setName("Sunny");
    Address address = new Address();
    address.setCity("Hyderabad");
    emp.setAddress(address);
    System.out.println(emp.getName() + " "+ emp.getAddress().getCity());//Sunny Hyderabad
```

Advantages of Aggregation:

Code reusability.

Composition: Composition is a restricted form of Aggregation in which two entities are highly dependent on each other. It represents part-of relationship. When there is a composition between two entities, the composed object cannot exist without the other entity.

Here if Car and Engine are highly dependent. Here there is no existence of car without engine vice versa if we destroy the car engine also to be destroyed.



```
class Vehicle{
    String name;
    Engine engine;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Engine getEngine() {
        return engine;
    }
    public void setEngine(Engine engine) {
        this.engine = engine;
    }
}

class Engine{
    int engineId;

    public int getEngineId() {
        return engineId;
    }
    public void setEngineId(int engineId) {
        this.engineId = engineId;
    }
}
public class Composition {
    public static void main(String[] args) {
        Vehicle vehicle = new Vehicle();
        vehicle.setName("Car");
        Engine engine = new Engine();
        engine.setEngineId(32242525);
        vehicle.setEngine(engine);

        System.out.println(vehicle.getName() + " " + vehicle.getEngine().getEngineId());//Car 32242525
```

44. Difference between Association and Inheritance?

Inheritance: It is process of acquiring the parent class fields, methods into sub class. It can be of different types. **It is is-a relationship.**

Single

Multi-Level

Hierarchical

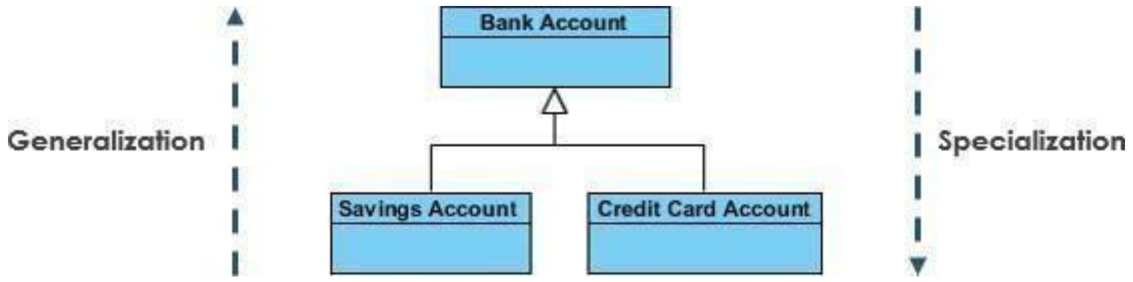
Multiple

Composition: The composition also provides code reusability but the difference here is we do not extend the class for this. . **It is has-a relationship.**

45. What is the difference between Generalization and Realization and Dependency?

Generalization: It is a mechanism for combining similar classes of objects into a single, more general class. Generalization identifies commonalities among a set of entities. The commonality may be of attributes, behavior, or both.

Specialization: It is the reverse process of Generalization means creating new sub-classes from an existing class.
For Example, a Bank Account is of two types - Savings Account and Credit Card Account. Savings Account and Credit Card Account inherit the common/ generalized properties like Account Number, Account Balance, etc. from a Bank Account and also have their specialized properties like unsettled payment etc.



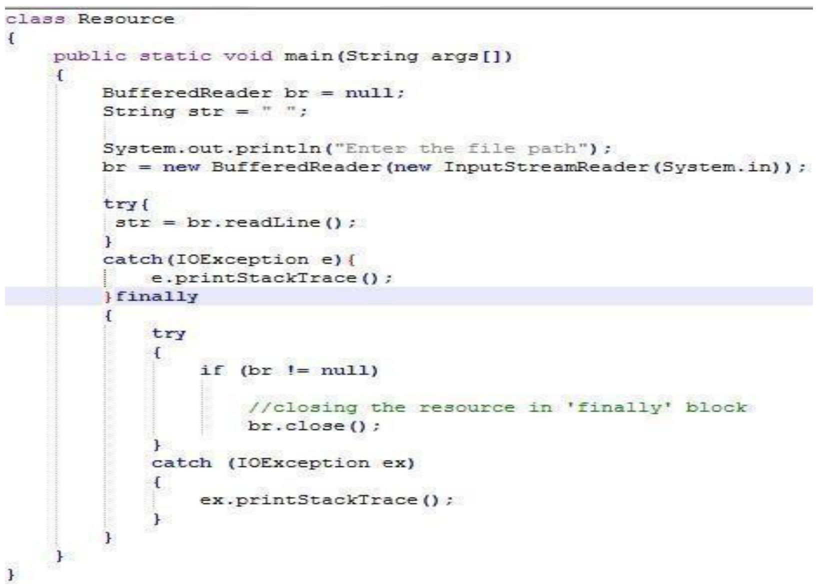
46. Difference between Generalization and Inheritance?

When we implement Generalization in a programming language, it is often called Inheritance instead. Generalization and inheritance are the same.

47. Explain about try-with-resource?

Try-with-resource is introduced in Java 7. Before Java 7 the closing of resources was done using the finally block.

Before Java 7:



```
class Resource
{
    public static void main(String args[])
    {
        BufferedReader br = null;
        String str = " ";

        System.out.println("Enter the file path");
        br = new BufferedReader(new InputStreamReader(System.in));

        try{
         str = br.readLine();
        }
        catch(IOException e){
            e.printStackTrace();
        }finally
        {
            try
            {
                if (br != null)

                    //closing the resource in 'finally' block
                    br.close();
            }
            catch (IOException ex)
            {
                ex.printStackTrace();
            }
        }
    }
}
```

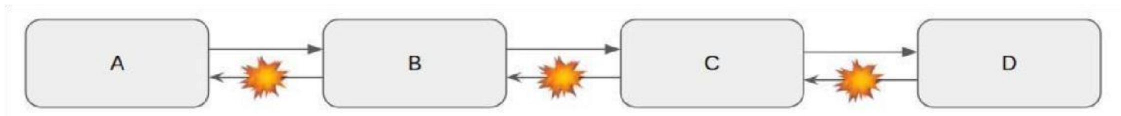
In Java, the try-with-resources statement is a try statement that declares one or more resources. The resource is as an object that must be closed after finishing the program. The trywithresources statement ensures that each resource is closed at the end of the statement execution.

```
import java.io.FileOutputStream;
public class TryWithResources {
public static void main(String args[]){
        // Using try-with-resources
try(FileOutputStream fileOutputStream =newFileOutputStream("/java7-new-features/src/abc.txt")){
String msg = "Welcome to javaTpoint!";
byte byteArray[] = msg.getBytes(); //converting string into byte array
fileOutputStream.write(byteArray);
System.out.println("Message written to file successfuly!");
}catch(Exception exception){
       System.out.println(exception);
}
}
}
```

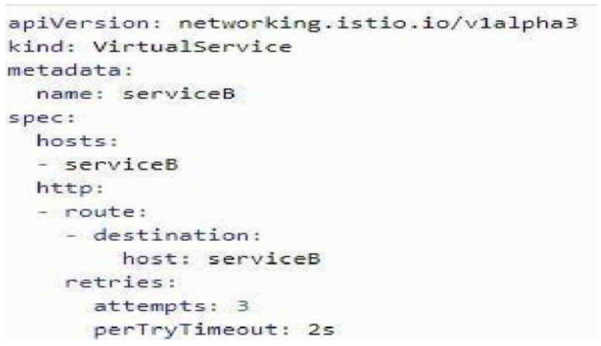
48. How to implement Retry in micro services?

To achieve the Resiliency in micro services spring providing one design pattern Retry Patter.

In Microservice architecture, when there are multiple services (A, B, C & D), one service (A) might depend on the other service (B) which in turn might depend on C and so on. Sometimes due to some issue, Service D might not respond as expected. Service D might have thrown some exception like OutOfMemory Error or Internal Server Error.

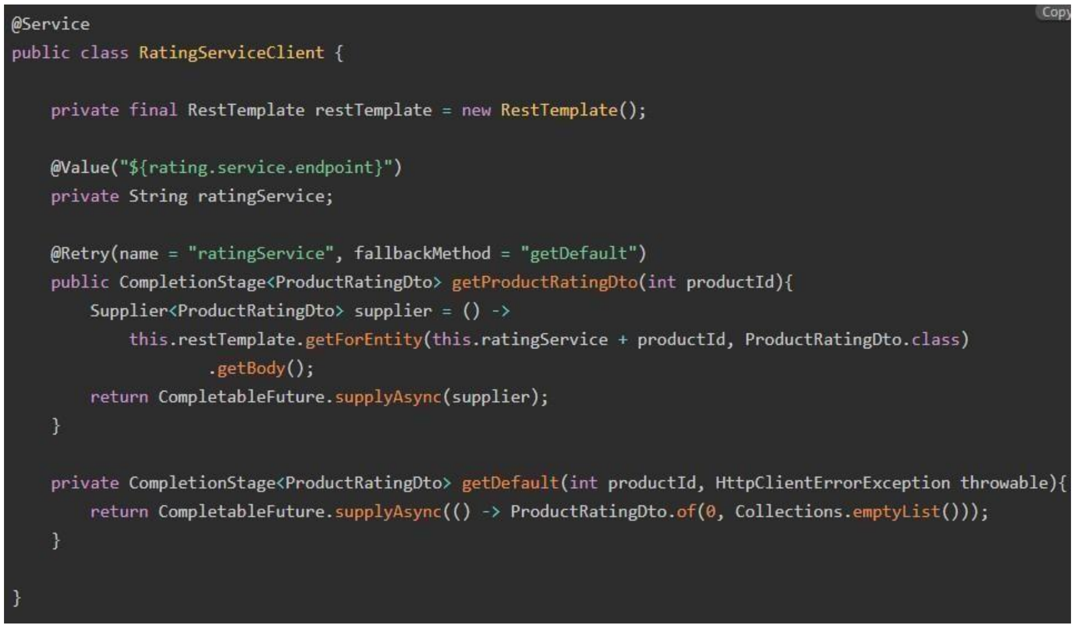


We need to add the below properties in yml file of the consumer application,



```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: serviceB
spec:
  hosts:
  - serviceB
  http:
  - route:
    - destination:
        host: serviceB
    retries:
      attempts: 3
      perTryTimeout: 2s
```

In the Java code we need to add the below code.



```java
@Service
public class RatingServiceClient {

    private final RestTemplate restTemplate = new RestTemplate();

    @Value("${rating.service.endpoint}")
    private String ratingService;

    @Retry(name = "ratingService", fallbackMethod = "getDefault")
    public CompletionStage<ProductRatingDto> getProductRatingDto(int productId){
        Supplier<ProductRatingDto> supplier = () ->
            this.restTemplate.getForEntity(this.ratingService + productId, ProductRatingDto.class)
                    .getBody();
        return CompletableFuture.supplyAsync(supplier);
    }

    private CompletionStage<ProductRatingDto> getDefault(int productId, HttpClientErrorException throwable){
        return CompletableFuture.supplyAsync(() -> ProductRatingDto.of(0, Collections.emptyList()));
    }

}
```

49. Difference between Retry and Circuit Breaker Patters?

The purpose of the Circuit Breaker pattern is different than the Retry pattern. The Retry pattern enables an application to retry an operation in the expectation that it'll succeed. The Circuit Breaker pattern prevents an application from performing an operation that is likely to fail. An application can combine these two patterns by using the Retry pattern to invoke an operation through a circuit breaker. However, the retry logic should be sensitive to any exceptions returned by the circuit breaker and abandon retry attempts if the circuit breaker indicates that a fault is not transient.

50. How PCF works internally when deploying Spring Boot project?

When you run cf push -p my/cool/file.jar (or even file.war), the cf cli extracts everything from that archive and pushes it up to CF. CF stores your app files & then your app is staged.

During staging, the Java build pack runs. It looks at all the files that were pushed & tries to determine what to do with them. It knows how to handle several different types of apps [1], including both standard WAR files & Spring Boot apps. The build pack will check your app to see if its one of the supported types in order [2] and will select the first match.

After selecting the type of app, it runs through and installs what is necessary to run your app.

For a Spring Boot app, that's basically just the JVM. For a WAR file, it installs Tomcat & a JVM. In addition, it writes out the configuration & start up commands necessary for CF to launch your app.

At this point staging is complete and you have what is called a "droplet". If you have any additional questions on the staging workflow, read here for more details [3].

At this point, the app would be started. The platform takes the droplet that was created and executes the command specified by the build pack to start the app [4]. If all goes well, your app will then be up and running on CF.

51. What are all the design patterns followed by Java 8?

Abstract Factory: A lambda that conforms to some interface and returns a new object.

Adapter: A lambda that calls a function with a different signature on some other object.

Chain of responsibility: A lambda that may or may not delegate to another lambda, which may or may not delegate to another lambda, ad infinitum. (How to get the 'next' reference into the lambda is left as an exercise for the reader.)

Command: Otherwise known as: a lambda! (Assuming you're not planning on implementing undo. But then you just want a tuple of lambdas, don't you?)

Decorator: A lambda that calls another lambda with the same signature but changes the arguments on the way in, or the result on the way out, or performs some extra action. (Assuming the decorated object has one public function.)

Iterator: Close (though not identical) to forEach(lambda). More specific functions like map(lambda), filter(lambda), flatMap(lambda), foldLeft/Right(lambda), reduceLeft/Right(lambda), etc. cater for the majority of Iterator's use in modern Java.

Observer: Give some other object a lambda to call when something happens in the future. (Assuming the Observer interface has a single function.)

Strategy: Choose from a family of lambdas with identical signatures at runtime.

Template method: Replace the abstract method polymorphism with composition, passing lambdas into the constructor.

52. How to convert from Monolith to Micro Services?

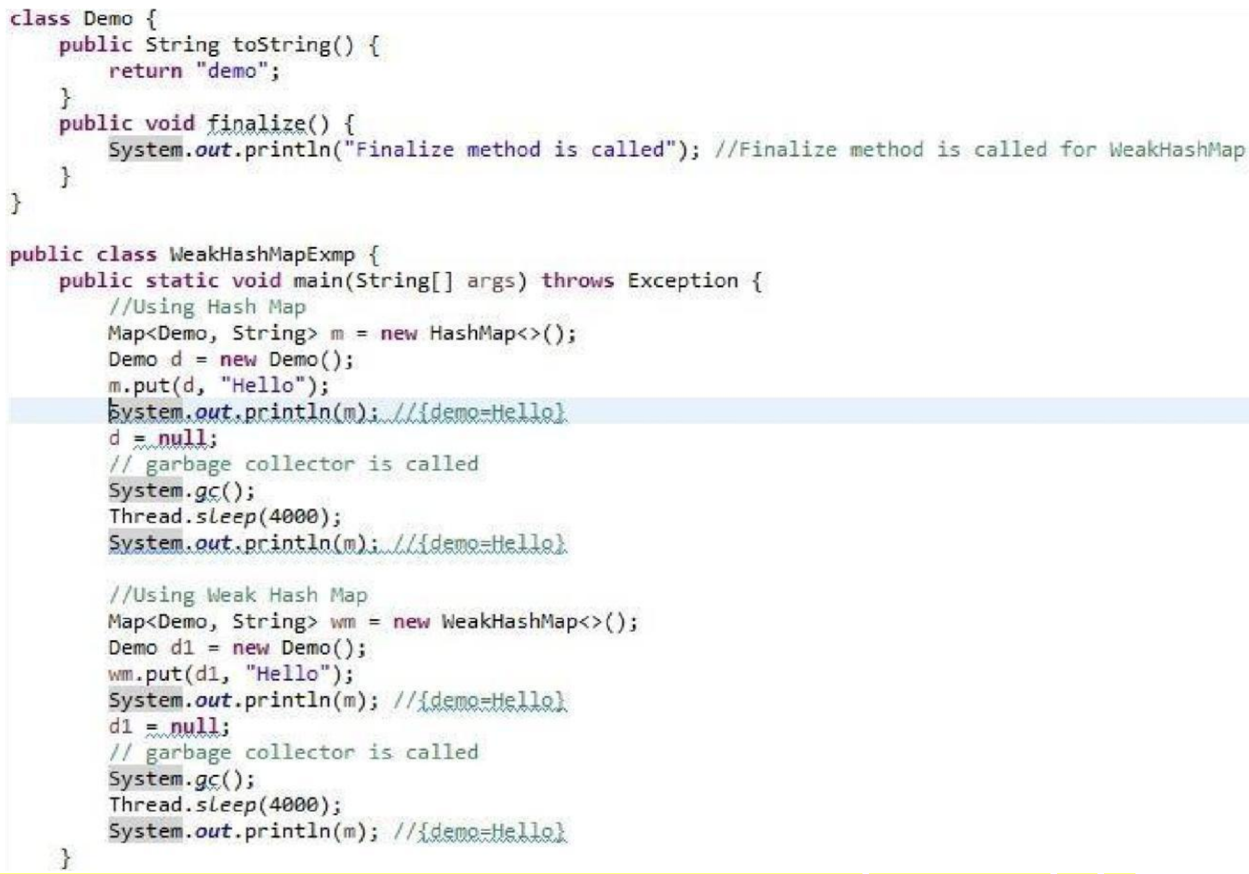
In my project we have all functionalities related Login, Register, Forgot password and Forgot user id in in single module.

First we analyzed all dependencies related to Registration functionality. Then created micro service for it with only registration functionality and database for it.

Then we created one proxy and started render the requests to registration micro service. We make sure it is working as expected. Then removed the actual references of registration functionality in monolith application.

Next step we divided remaining functionalities Login, Forgot password and Forgot user id as separate micro services and removed actual references to monolith applications by following the above approach.

53. Write an example for WeakHashMap?

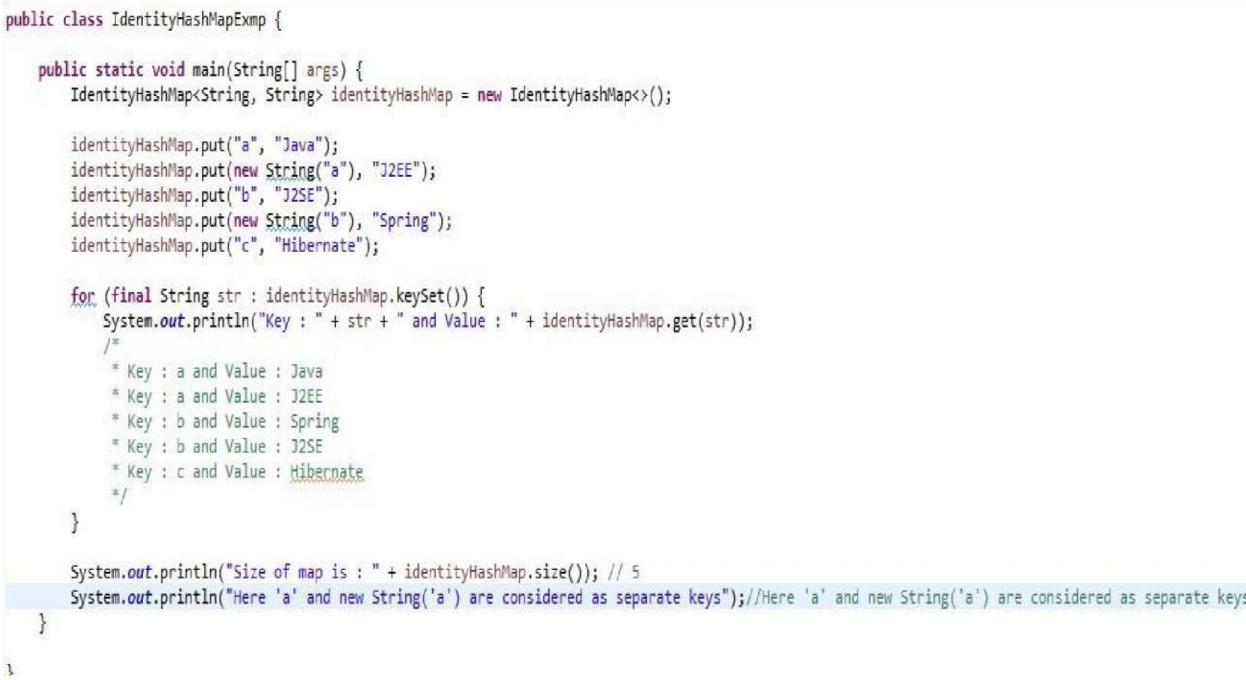


```java
class Demo {
    public String toString() {
        return "demo";
    }
    public void finalize() {
        System.out.println("Finalize method is called"); //Finalize method is called for WeakHashMap
    }
}

public class WeakHashMapExmp {
    public static void main(String[] args) throws Exception {
        //Using Hash Map
        Map<Demo, String> m = new HashMap<>();
        Demo d = new Demo();
        m.put(d, "Hello");
        System.out.println(m); //{demo=Hello}
        d = null;
        // garbage collector is called
        System.gc();
        Thread.sleep(4000);
        System.out.println(m); //{demo=Hello}

        //Using Weak Hash Map
        Map<Demo, String> wm = new WeakHashMap<>();
        Demo d1 = new Demo();
        wm.put(d1, "Hello");
        System.out.println(m); //{demo=Hello}
        d1 = null;
        // garbage collector is called
        System.gc();
        Thread.sleep(4000);
        System.out.println(m); //{demo=Hello}
    }
```

In the above example finalize method is called when we invoke the operations on WeakHashMap not on HashMap.

54. Write an example for IdentityHashMap?

As we know that IdentityHashMap is a HashTable based implementation of Map Interface. Normal HashMap compares keys using '.equals' method. But Identity HashMap compares its keys using '==' operator. Hence 'a' and new String('a') are considered as 2 different keys. The initial size of Identity HashMap is 21 while the initial size of normal HashMap is 16.



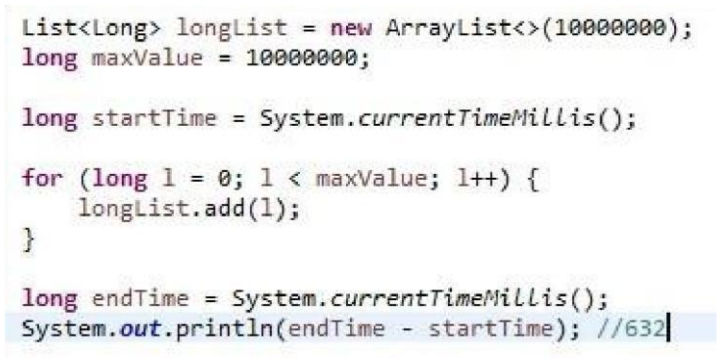
```
public class IdentityHashMapExmp {

    public static void main(String[] args) {
        IdentityHashMap<String, String> identityHashMap = new IdentityHashMap<>();

        identityHashMap.put("a", "Java");
        identityHashMap.put(new String("a"), "J2EE");
        identityHashMap.put("b", "J2SE");
        identityHashMap.put(new String("b"), "Spring");
        identityHashMap.put("c", "Hibernate");

        for (final String str : identityHashMap.keySet()) {
            System.out.println("Key : " + str + " and Value : " + identityHashMap.get(str));
            /*
             * Key : a and Value : Java
             * Key : a and Value : J2EE
             * Key : b and Value : Spring
             * Key : b and Value : J2SE
             * Key : c and Value : Hibernate
             */
        }

        System.out.println("Size of map is : " + identityHashMap.size()); // 5
        System.out.println("Here 'a' and new String('a') are considered as separate keys");//Here 'a' and new String('a') are considered as separate keys
    }
}
```

55. I have 20 GB data in database which collection you will use store it and identify the common strings and its count in it?

Here ArrayList is the better option compared to LinkedList. Because of the following reason,

Regarding Space: Well ArrayList will take less memory as compared to LinkedList because we all know that LinkedList maintain doubly Linked-List so LinkedList require extra space to keep the next and previous elements.

Regarding Time: Which will again reduce the insertion time of large number of elements. At the time of ArrayList declaration, declared with initial capacity because you already know how much element you want to store.



```
List<Long> longList = new ArrayList<>(10000000);
long maxValue = 10000000;

long startTime = System.currentTimeMillis();

for (long l = 0; l < maxValue; l++) {
    longList.add(l);
}

long endTime = System.currentTimeMillis();
System.out.println(endTime - startTime); //632
```

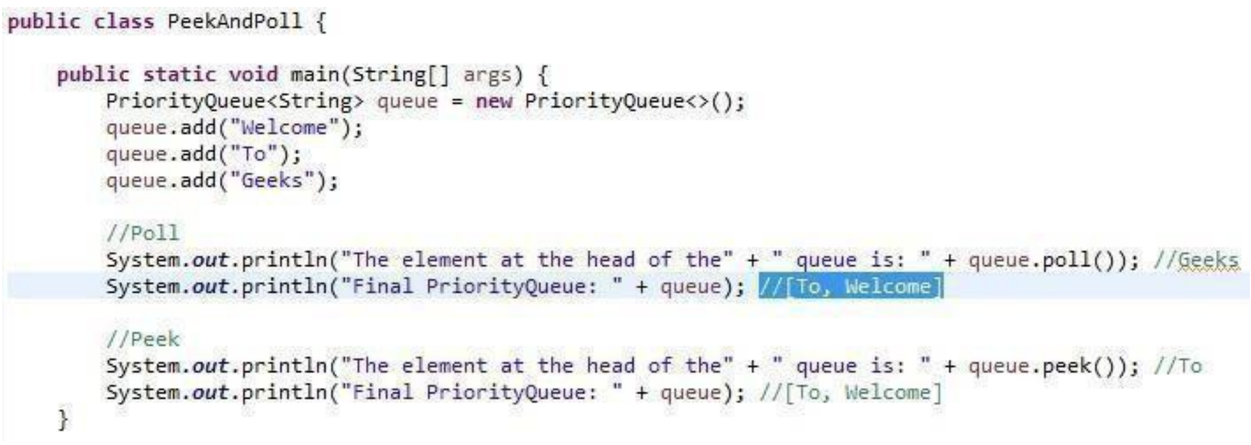
56. What is meant by bounded context in Micro Services?

A Bounded context should be an independent domain, if the system is correctly designed; in reality, when things are not done correctly, a Bounded context is larger than a domain. In large enterprises, some developers create objects (models) that try to capture all the behavior related to some term. For example, Product in a Shop. This term is very broad. The Product from the online-shop and the Product from the inventory system are not one and the same thing, although they may seem that way. In this case, the online-shop should be bounded context and the inventory should be a different one.

You can then have a separate microservice corresponding to each Bounded Context. For Example: If you in Insurance Domain, then you can have bounded context like Customer, Quote, Policy etc. and can have a micro-service for each of them.

57. Difference between Peek and Poll?

The java.util.PriorityQueue.poll() method in Java is used to retrieve or fetch and remove the first element of the Queue or the element present at the head of the Queue. The peek() method only retrieved the element at the head but the poll() also removes the element along with the retrieval. It returns NULL if the queue is empty.



```
public class PeekAndPoll {

    public static void main(String[] args) {
        PriorityQueue<String> queue = new PriorityQueue<>();
        queue.add("Welcome");
        queue.add("To");
        queue.add("Geeks");

        //Poll
        System.out.println("The element at the head of the" + " queue is: " + queue.poll()); //Geeks
        System.out.println("Final PriorityQueue: " + queue); //[To, Welcome]

        //Peek
        System.out.println("The element at the head of the" + " queue is: " + queue.peek()); //To
        System.out.println("Final PriorityQueue: " + queue); //[To, Welcome]
    }
```

58. Explain about SOLID principles?

S – Single Responsibility
O – Open Close Principle
L - Liskov Substitution
I – Interface Segregation
D – Dependency Inversion

Single Responsibility: One class should have one and only one responsibility.

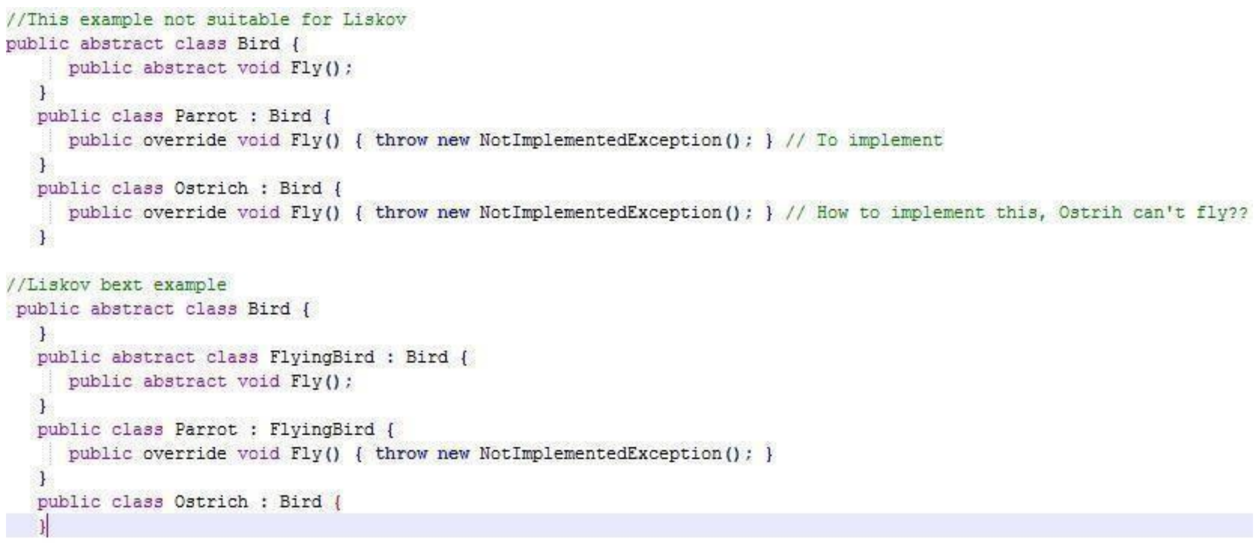
Example: Person class can have person's related data and Account must have account related data.

Open Close Principle: Software components should be open for extension, but closed for modification. It means that the application classes should be designed in such a way that whenever fellow developers want to change the flow of control in specific conditions in application, all they need to extend the class and override some functions and that's it.

Example: For example, spring framework has class DispatcherServlet. This class acts as a front controller for String based web applications. To use this class, we are not required to modify this class. All we need is to pass initialization parameters and we can extend its functionality the way we want.

Liskov Substitution: Objects of a superclass should be replaceable with objects of its subclasses without breaking the system.

In other words, if class A is a subtype of class B, then we should be able to replace B with A without interrupting the behavior of the program.



```
//This example not suitable for Liskov
public abstract class Bird {
        public abstract void Fly();
    }
    public class Parrot : Bird {
        public override void Fly() { throw new NotImplementedException(); } // To implement
    }
    public class Ostrich : Bird {
        public override void Fly() { throw new NotImplementedException(); } // How to implement this, Ostrih can't fly??
    }

//Liskov bext example
 public abstract class Bird {
    }
    public abstract class FlyingBird : Bird {
        public abstract void Fly();
    }
    public class Parrot : FlyingBird {
        public override void Fly() { throw new NotImplementedException(); }
    }
    public class Ostrich : Bird {
    }
```

Interface Segregation Principle: Clients should not be forced to implement unnecessary methods which they will not use

Example: Take an example. Developer Alex created an interface Reportable and added two methods generateExcel() and generatedPdf(). Now client 'A' wants to use this interface but he intends to use reports only in PDF format and not in excel. Will he be able to use the functionality easily?

Solution is to create two interfaces by breaking the existing one. They should be like PdfReportable and ExcelReportable. This will give the flexibility to users to use only the required functionality only.

Dependency Inversion Principle: We should design our software in such a way that various modules can be separated from each other using an abstract layer to bind them together.

Example: In the spring framework, all modules are provided as separate components which can work together by simply injecting dependencies in other modules. This dependency is managed externally in XML files.

59. How ConcurrentHashMap will work internally?

The ConcurrentHashMap class is introduced in JDK 1.5 belongs to java.util.concurrent package.

The underlined data structure for ConcurrentHashMap is Hashtable.

ConcurrentHashMap class is thread-safe i.e. multiple threads can operate on a single object without any complications.

At a time any number of threads are applicable for a read operation without locking the ConcurrentHashMap object which is not there in HashMap.

In ConcurrentHashMap, the Object is divided into a number of segments according to the concurrency level.

The default concurrency-level of ConcurrentHashMap is 16.

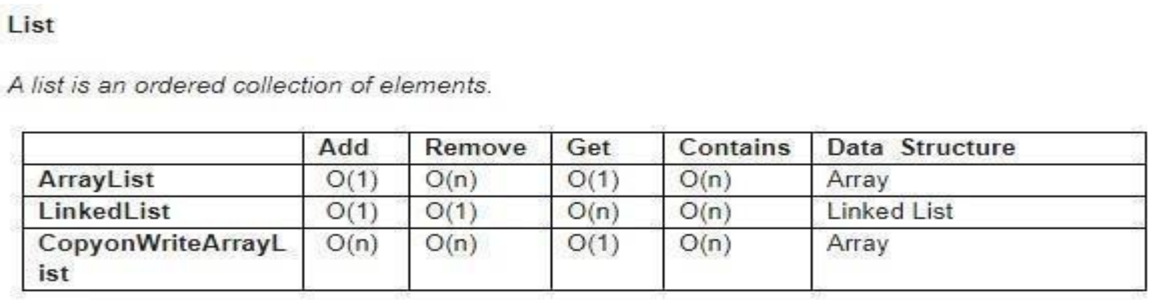
In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updated in the object, the thread must lock the particular segment in which the thread wants to operate. This type of locking mechanism is known as Segment locking or bucket locking. Hence at a time, 16 update operations can be performed by threads.

Inserting null objects is not possible in ConcurrentHashMap as a key or value.

60. Difference between HashTable, ConcurrentHashMap and SynchronizedMap?

ConcurrentHashMap	**HashTable**	**SynchronizedMap**
We will get thread safety without locking the total map object just with a bucket level lock.	We will get thread safety by locking the whole map object	We will get thread safety by locking the whole map object.
At a time multiple threads are allowed to operate on map objects safely.	At a time one thread is allowed to operate on a map object.	At a time only one thread is allowed to perform any operation on a map object.
Iterator of ConcurrentHashMap is fail-safe and won't raise ConcurrentModificationExcepti on	Iterator of HashTable is fail-fast and it will raise ConcurrentModificationExcepti on	Iterator of SynchronizedMap is fail-fast and it will raise ConcurrentModificationExcepti on
Null is not allowed for both keys and values.	Null is not allowed for both keys and values	Null is allowed for both keys and values.

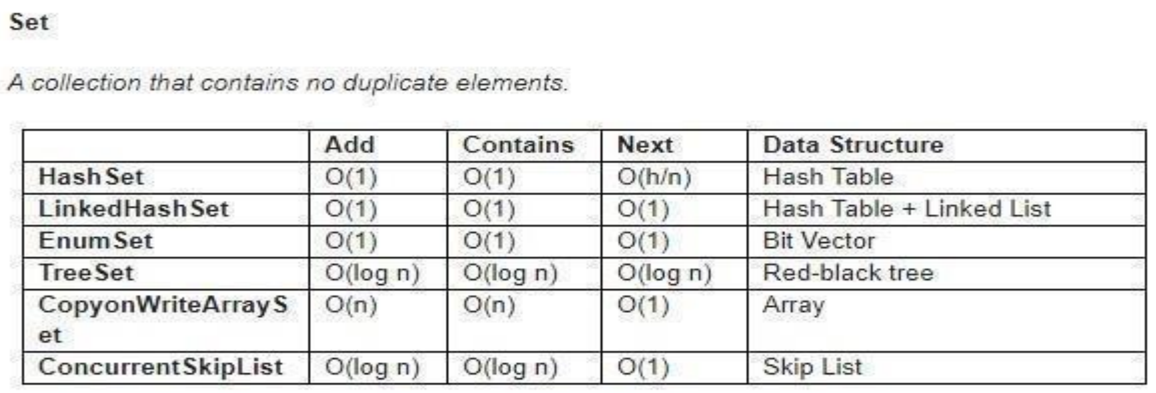
61. Explain time and space complexities of ArrayList, HashSet and HashMap?



List

A list is an ordered collection of elements.

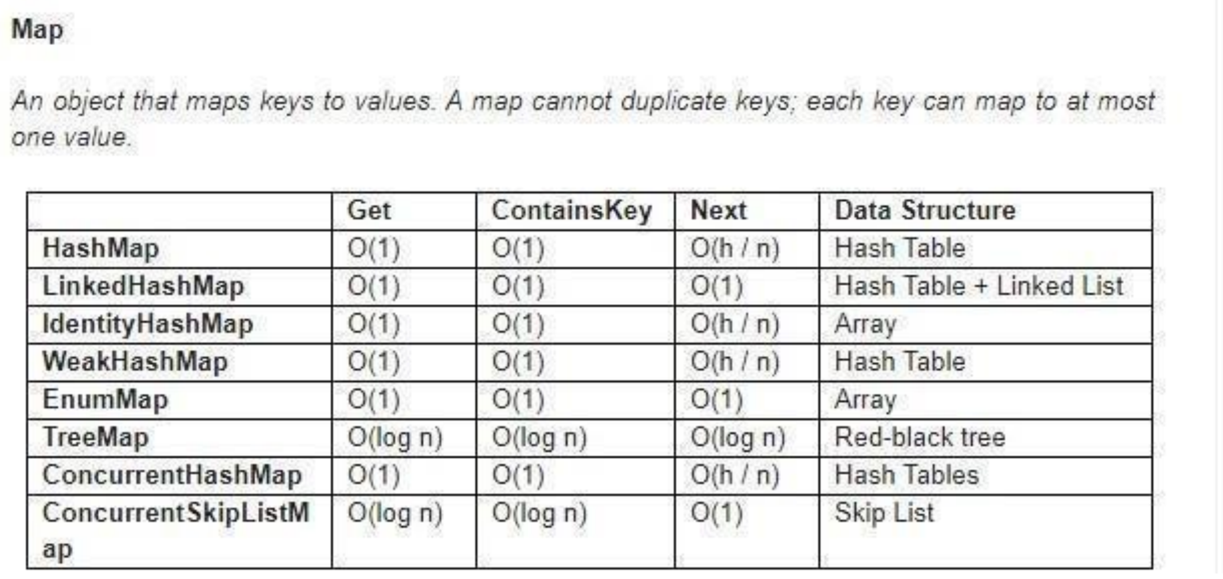
	Add	Remove	Get	Contains	Data Structure
ArrayList	O(1)	O(n)	O(1)	O(n)	Array
LinkedList	O(1)	O(1)	O(n)	O(n)	Linked List
CopyonWriteArrayList	O(n)	O(n)	O(1)	O(n)	Array



Set

A collection that contains no duplicate elements.

	Add	Contains	Next	Data Structure
HashSet	O(1)	O(1)	O(h/n)	Hash Table
LinkedHashSet	O(1)	O(1)	O(1)	Hash Table + Linked List
EnumSet	O(1)	O(1)	O(1)	Bit Vector
TreeSet	O(log n)	O(log n)	O(log n)	Red-black tree
CopyonWriteArraySet	O(n)	O(n)	O(1)	Array
ConcurrentSkipList	O(log n)	O(log n)	O(1)	Skip List



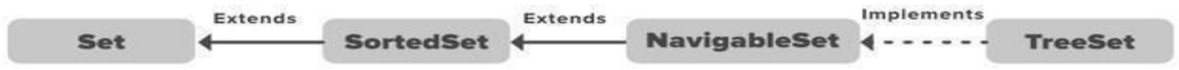
Map

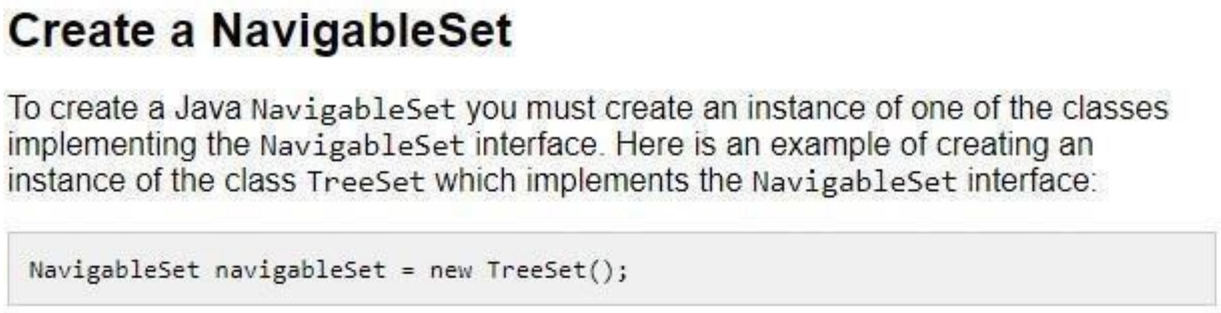
An object that maps keys to values. A map cannot duplicate keys; each key can map to at most one value.

	Get	ContainsKey	Next	Data Structure
HashMap	O(1)	O(1)	O(h / n)	Hash Table
LinkedHashMap	O(1)	O(1)	O(1)	Hash Table + Linked List
IdentityHashMap	O(1)	O(1)	O(h / n)	Array
WeakHashMap	O(1)	O(1)	O(h / n)	Hash Table
EnumMap	O(1)	O(1)	O(1)	Array
TreeMap	O(log n)	O(log n)	O(log n)	Red-black tree
ConcurrentHashMap	O(1)	O(1)	O(h / n)	Hash Tables
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	Skip List

62. Explain about NavigableSet and NavigableMap?

NavigableSet: It is an interface. A SortedSet extended with navigation methods reporting closest matches for given search targets. Methods lower, floor, ceiling, and higher return elements respectively less than, less than or equal, greater than or equal, and greater than a given element, returning null if there is no such element. A NavigableSet may be accessed and traversed in either ascending or descending order. The **descendingSet** method returns a view of the set with the senses of all relational and directional methods inverted. The performance of ascending operations and views is likely to be faster than that of descending ones. This interface additionally defines methods **pollFirst** and **pollLast** that return and remove the lowest and highest element, if one exists, else returning null. Methods **subSet**, **headSet**, and **tailSet** differ from the like-named SortedSet methods in accepting additional arguments describing whether lower and upper bounds are inclusive versus exclusive. Subsets of any NavigableSet must implement the NavigableSet interface.



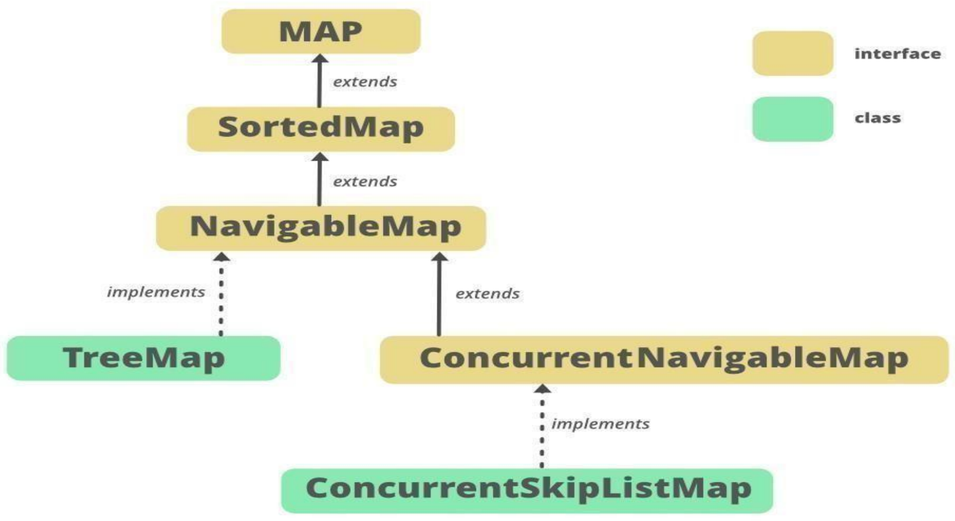


Create a NavigableSet

To create a Java `NavigableSet` you must create an instance of one of the classes implementing the `NavigableSet` interface. Here is an example of creating an instance of the class `TreeSet` which implements the `NavigableSet` interface:

```
NavigableSet navigableSet = new TreeSet();
```

NavigableMap: The NavigableMap interface is a member of the Java Collection Framework. It belongs to java.util package and It is an extension of SortedMap which provides convenient navigation methods like **lowerKey, floorKey, ceilingKey and higherKey**, and along with this popular navigation method.



Example:
NavigableMap<Integer, String> nmap = new TreeMap<Integer, String>();

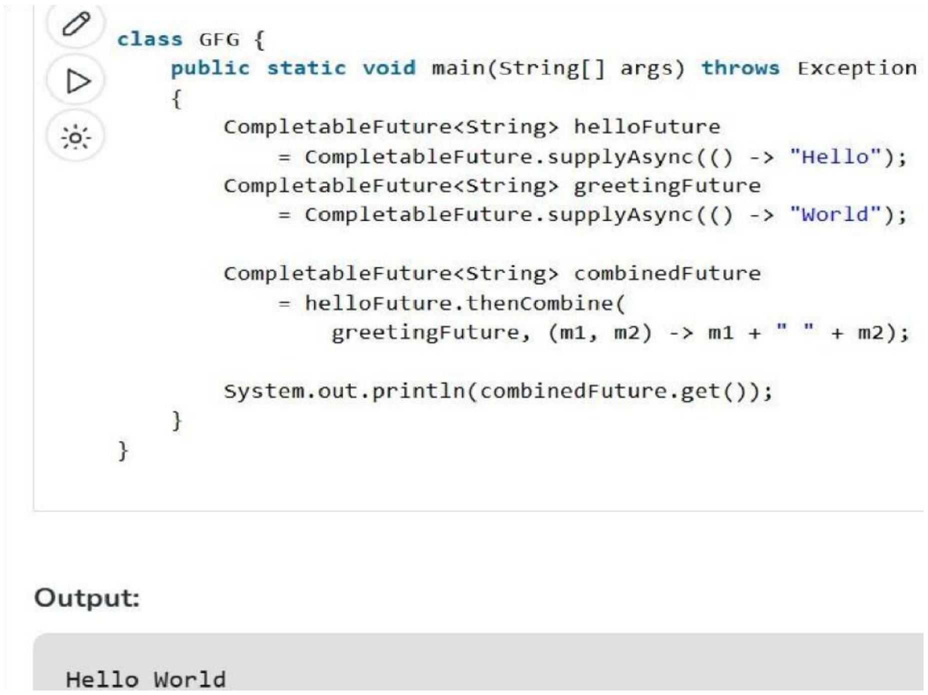
63. Difference between CountDownLatch & CyclicBarrier ?

CyclicBarrier: The CyclicBarrier class in Java is designed to allow a group of threads to wait for each other to reach a common point, where they can synchronize and proceed together. The barrier is cyclic because it can be reused after all threads have reached it.

CountDownLatch: The CountDownLatch class in Java is used to make a thread wait until the count of CountDownLatch reaches zero. It's a one-time use synchronization tool where the count cannot be reset, making it suitable for scenarios where a specific number of tasks need to be completed before a process continues.

63. Explain about CompletableFuture in java?

CompletableFuture provides a powerful and flexible way to write asynchronous, non-blocking code in Java. It was introduced in Java 8 . We can use it to compose multiple asynchronous operations, handle errors and exceptions, and combine multiple CompletableFutures into one. By using CompletableFuture, we can write more efficient and scalable code that can take advantage of multi-core processors and handle complex asynchronous workflows.



```
class GFG {
    public static void main(String[] args) throws Exception
    {
        CompletableFuture<String> helloFuture
            = CompletableFuture.supplyAsync(() -> "Hello");
        CompletableFuture<String> greetingFuture
            = CompletableFuture.supplyAsync(() -> "World");

        CompletableFuture<String> combinedFuture
            = helloFuture.thenCombine(
                greetingFuture, (m1, m2) -> m1 + " " + m2);

        System.out.println(combinedFuture.get());
    }
}
```

Output:

```
Hello World
```

64. Difference between Future and CompletableFuture?

Future and CompletableFuture are both abstractions for representing a result that will be available in the future, but there are some important differences between them.

Blocking vs non-blocking: One of the key differences between Future and CompletableFuture is that Future is a blocking API, whereas CompletableFuture is non-blocking. With a Future object, you must call the get() method to retrieve the result, but this method blocks until the result is available. In contrast, with a CompletableFuture object, you can use various non-blocking methods to retrieve the result, such as thenApply(), thenAccept(), or join().

Composition: CompletableFuture provides a more powerful composition API than Future. With Future, it is difficult to chain multiple asynchronous operations together or to combine the results of multiple operations. CompletableFuture, on the other hand, provides methods such as thenCompose(), thenCombine(), and allOf() that make it easy to compose multiple asynchronous operations and to handle their results in a non-blocking way.

Exception Handling: CompletableFuture provides better exception handling than Future. With Future, you can only check if the computation completed successfully or not. If an exception occurs during the computation, you have to catch it explicitly. In contrast, with CompletableFuture, you can handle exceptions in a more declarative way using methods like exceptionally() and handle().

Completion: With a Future object, there is no way to explicitly complete the future. Once you submit a task to an executor service and get a Future object in return, you can only wait for the task to complete. With CompletableFuture, you have more control over the completion of the future. You can complete it explicitly by calling complete(), completeExceptionally(), or cancel() methods.

65. Explain different methods of CompletableFuture?

- **supplyAsync(Supplier<U>):** Runs an asynchronous task that supplies a result.
- **runAsync(Runnable):** Runs an asynchronous task that doesn't return a result.
- **thenApply(Function<T, U>):** Transforms the result of a computation.

```
public class CompletableFutureExample {
public static void main(String[] args) {
    CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello")
.thenApply(result -> result + " World!");

    System.out.println(future.join()); // Output: Hello World!
  }
}
```

- **thenCombine(CompletableFuture<U>, BiFunction<T, U, V>):** Combines the results of two CompletableFutures.

```
public class CompletableFutureExample {
public static void main(String[] args) {
    CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> "Hello");
    CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() ->
"World");

    CompletableFuture<String> combinedFuture = future1.thenCombine(future2, (a, b) -
> a + " " + b);

    System.out.println(combinedFuture.join()); // Output: Hello World
  }
}
```

- **exceptionally(Function<Throwable, U>):** Handles exceptions in the computation.

```
public class CompletableFutureExample {
public static void main(String[] args) {
    CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
      return 10 / 0; // Division by zero
    }).exceptionally(ex -> {
      System.out.println("Exception: " + ex.getMessage());
return 0;
    });

    System.out.println(future.join()); // Output: Exception: / by zero
  } }
```

65. Difference between sharding and partitioning?

Sharding and partitioning are both methods of dividing data across multiple databases or storage systems to improve performance and manageability. However, they differ in their approach and use cases.

Sharding:

Definition: Sharding involves breaking up a large dataset into smaller, more manageable pieces called shards. Each shard is a separate database or storage unit, and together they form a complete dataset.
Use Case: Sharding is often used in distributed database systems where the dataset is too large to be handled by a single database. It helps improve performance by distributing the load across multiple servers. **Example**: Imagine a social media application with millions of users. Each user's data can be stored in a different shard based on their user ID, ensuring that no single shard becomes too large or overloaded.

Partitioning:

Definition: Partitioning involves dividing a database table into smaller, more manageable pieces called partitions. Each partition is stored within the same database but can be managed and queried separately. **Use Case**: Partitioning is often used to improve query performance and manage large tables. It helps in organizing data logically and reducing the amount of data that needs to be scanned during queries.
Example: In an e-commerce application, an orders table can be partitioned by order date. This way, queries for recent orders only need to scan a small, relevant portion of the table rather than the entire dataset.

65. Difference between supplyAsync() and runAsync()?

***runAsync()* is a method used to execute a task asynchronously that doesn’t produce a result**. It’s suitable for fire-and-forget tasks where we want to execute code asynchronously without waiting for a result. For example, logging, sending notifications, or triggering background tasks. **It returns a *CompletableFuture<Void>* and is useful for scenarios where the focus is on the completion of a task rather than obtaining a specific result.**

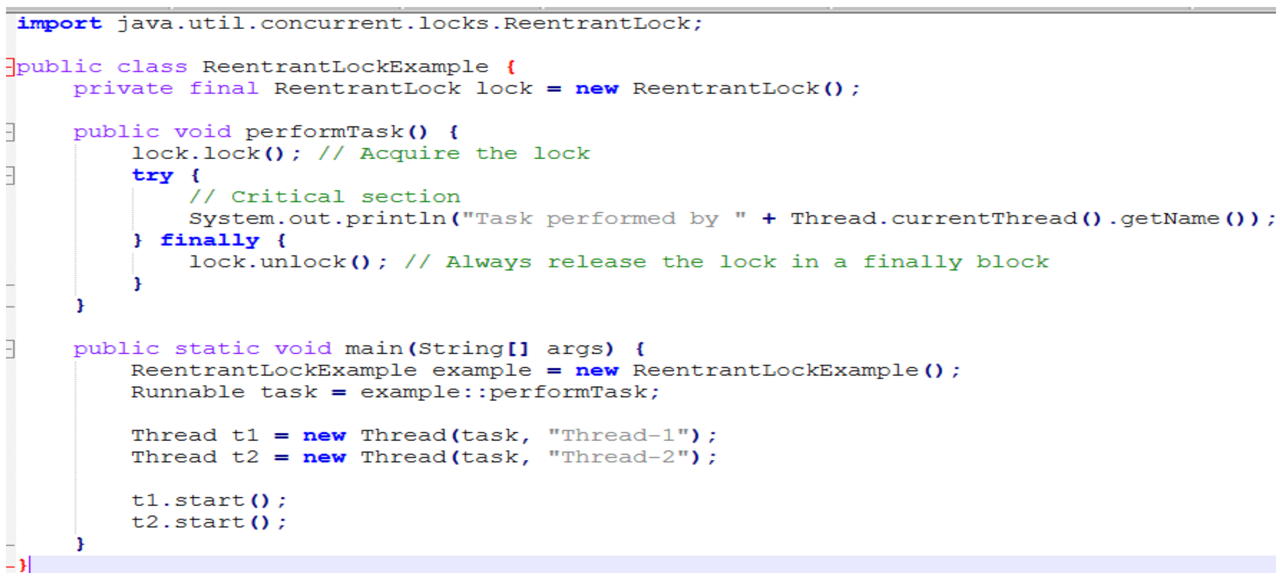
```
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
  // Perform non-result producing task
  System.out.println("Task executed asynchronously");
});
```

On the other hand, *supplyAsync()* is a method used to asynchronously execute a task that produces a result. It's ideal for tasks that require a result for further processing. For example, fetching data from a database, making an API call, or performing a computation asynchronously. **Subsequently, it returns a *CompletableFuture<T>*, where *T* is the type of the result produced by the task.**

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    // Perform result-producing task
    return "Result of the asynchronous computation";
});

// Get the result later
String result = future.get();
System.out.println("Result: " + result);
```

65. Explain about ReentrantLock ?

ReentrantLock is a mutual exclusion mechanism in Java that allows threads to reenter into a lock on a resource multiple times without a deadlock situation. It is a class in Java that provides a way to create a mutually exclusive locking mechanism for threads. It allows multiple threads to access a shared resource, but only one thread can access it at a time. It is a more advanced alternative to the synchronized keyword in Java.
ReentrantLock tracks a "hold count" which is a value that starts at 1 when a thread first locks the resource. Each time the thread re-enters the lock, the count is incremented. The count is decremented when the lock is released. Once the hold count reaches zero, the lock is fully released.



```
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockExample {
    private final ReentrantLock lock = new ReentrantLock();

    public void performTask() {
        lock.lock(); // Acquire the lock
        try {
            // Critical section
            System.out.println("Task performed by " + Thread.currentThread().getName());
        } finally {
            lock.unlock(); // Always release the lock in a finally block
        }
    }

    public static void main(String[] args) {
        ReentrantLockExample example = new ReentrantLockExample();
        Runnable task = example::performTask;

        Thread t1 = new Thread(task, "Thread-1");
        Thread t2 = new Thread(task, "Thread-2");

        t1.start();
        t2.start();
    }
}
```